National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

THE UNIVERSITY OF ALBERTA

Godel: A Prototype Prolog Programming Environment

by

Daniel Andrew Lanovaz

(C)

A Thesis

Submitted to the faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master Of Science

Department of Computing Science

Edmonton, Alberta

Fall 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45758-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Daniel Andrew Lanovaz**

TITLE OF THESIS: **Godel: A Prototype Prolog Programming Environment**

DEGREE: **Master Of Science**

YEAR THIS DEGREE GRANTED: **1988**

Permissions is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....Daniel Lanovaz.....

Permanent Address:

P.O. Box 53
Grand Centre, AB
Canada T0A 1T0

Date: October 6, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Godel: A Prototype Prolog Programming Environment** submitted by **Daniel Andrew Lanovaz** in partial fulfillment of the requirements for the degree of **Master Of Science**.

.....
Supervisor

.....

.....

.....

Date: October 6, 1988

To my parents

Abstract

Logic programming is a new and exciting declarative programming paradigm. Its most popular realization, Prolog, originally had several deficiencies ranging from relatively slow execution speeds on conventional processors to a lack of appropriate software engineering support. This thesis is a study of the Prolog language from the software engineering viewpoint. We outline the features added to the Prolog language to make it suitable for large software development. We also describe how these features are integrated into a prototype graphically oriented *Prolog Programming Environment*. In designing this environment we paid particular attention to user interface issues. The more notable components are a syntax directed editor that permits incremental construction of Prolog programs and provides static semantic checking, a predicate browser for modularized code, and a graphical trace and spy debugger. We also describe and comment on the object-oriented architecture for both the graphical interface and the language interpreter.

PREFACE

The most beautiful and deepest experience a man can have is the sense of the mysterious. It is the underlying principle of religion as well as of all serious endeavor in art and in science . . . . He who never had this experience seems to me, if not dead, then at least blind. The sense that behind anything that can be experienced there is a something that our mind cannot grasp and whose beauty and sublimity reaches us only indirectly and as feeble reflection, this is religiousness. *In this sense I am religious. To me it suffices to wonder at these secrets and to attempt humbly to grasp with my mind a mere image of the lofty structure of all that there is.*

- Albert Einstein, 1932

The Cosmic Computer

Earth, water, air and fire: the fundamental components which Aristotelian civilization believed to compose the universe. Fortunately, through the work of Copernicus, Kepler, Galileo, Newton, Einstein and others, the image of our universe has shifted dramatically from that primitive viewpoint. Their use of mathematics as a tool for describing the behavior of our universe by classifying the principle particles that form our very existence has led to the discovery of quantum theory and the mathematical nature of atomic events.

An interesting metaphor, created by Heinz Pagels, between this study of the physics of our universe and the modern computer is expressed in a term he coined, "the Cosmic Computer."[1] In this metaphor, the material things in the universe, the quantum particles, are the objects created, destroyed, and manipulated by the computer's

[1] Pagels, H. R., Perfect Symmetry: The Search For The Beginning Of Time, Bantam Books, 1985, p. 371-391

"hardware". The logical rules these particles obey, the *laws of nature*, are the "software", the executing "program" of the computer.

In this thesis, our universe is the computer. It is the environment which enforces the laws that these objects must obey. Our *laws of nature* are defined by a programming language based on mathematical logic while the creation of the objects (quantum) in the cosmic computer is modelled by the use of an object-oriented programming paradigm. The result is the description and analysis of a new world, a new programming environment for logic programming that is implemented using an object-oriented programming language.

Background

I assume the reader has some knowledge of conventional Prolog, logic programming, and mathematical logic. Three texts that describe conventional Prolog are [Sterling and Shapiro 1986], [Clark and McCabe 1984] and [Clocksin and Mellish 1981]. [Kowalski 1979] is a paper that concentrates on logic programming without discussing a particular implementation. Readings on mathematical logic are covered in [Kleene 1967] and [Hodges 1971].

Acknowledgements

I am extremely indebted to Dr. Duane Szafron. His earlier thoughts inspired many of the ideas developed in this thesis. Also, through his careful proofreading and insightful comments, I was able to produce a more polished and professional thesis. I am also grateful to Dr. Jia-Huai You, Dr. Randy Goebel, and Dr. Emil Girczyc for the time they took to read and comment on this work. Lastly, I wish to thank my loving parents who provided both encouragement and support during my Graduate studies.

Table Of Contents

Chapter **Page**

Chapter 1: Introduction 1
- 1.1. Logic Programming: A Brief History 3
- 1.2. Logic Programming Environments: The New Wave 6
- 1.3. Thesis Overview 8

Chapter 2: Prolog Programming Environment Architecture 10
- 2.1. Programming Environments 10
- 2.2. Prolog Environment Architecture 13
  - 2.2.1. Central Repository 15
  - 2.2.2. Modularization 16
  - 2.2.3. Typing 19
  - 2.2.4. Open World Logic Programming 23
- 2.3. Overview of Godel's User Interface 25
  - 2.3.1. User Interface Paradigm 25
  - 2.3.2. Interface Components 26
    - 2.3.2.1. Dialogs and Alerts 27
    - 2.3.2.2. Language Directed Editing 28
    - 2.3.2.3. Graphical Clausal Representations 31
    - 2.3.2.4. In-line Dialogs 34
    - 2.3.2.5. Code Browsing 35
    - 2.3.2.6. Procedural Debugging 37
      - 2.3.2.6.1. Debugging Models 38
      - 2.3.2.6.2. Breakpoints 40
      - 2.3.2.6.3. Execution Visualization 41
- 2.4. Summary 42

Chapter 3: Prolog Development Environment Design Details 43
- 3.1. Preliminary Definitions 43
  - 3.1.1. Herbrand Interpretations 43
  - 3.1.2. BNF Notation 45
- 3.2. Modularization in Logic Programming 46
  - 3.2.1. Module Glue 47
  - 3.2.2. Non-logical Predicates 49
    - 3.2.2.1. Assert and Retract 50
    - 3.2.2.2. The Meta-predicate Prove 50
    - 3.2.2.3. All Solution Predicates 51
    - 3.2.2.4. Negation as Failure 52
- 3.3. Declarations 52
  - 3.3.1. Predicate and Function Declarations 54
  - 3.3.2. External Declarations 55
  - 3.3.3. Type Declarations 56
  - 3.3.4. Polymorphic Types 59
  - 3.3.5. Overloading 60
  - 3.3.6. Additional Type Declarations 60
- 3.4. Typing Extensions 62
- 3.5. Static Semantic Errors 64
- 3.6. Summary 65

Chapter 4: Object Oriented Environment Design 66
4.1. Object-Oriented Programming 67
4.2. Guide Architecture 70
4.2.1. Class Hierarchy 70
4.2.1.1. Dependents 71
4.2.2. The Interface 72
4.2.2.1. MVC Paradigm 72
4.2.2.2. Structures 73
4.2.2.3. Structure Browsers 73
4.2.3. Symbol Table 74
4.2.4. Parse Tree 75
4.2.5. Parser 76
4.3. Godel Architecture 76
4.3.1. Class Hierarchy 77
4.3.2. Symbol Table 77
4.3.3. Parse Tree 78
4.3.3.1. Parse Tree Optimizations 81
4.3.4. Parser 83
4.4. Summary 83

Chapter 5: Prolog Interpretation 85
5.1. Unification 86
5.2. Interpreting Algorithm 88
5.3. Memory Organization 90
5.4. Interpreter Optimizations 93
5.5. Summary 94

Chapter 6: Object-Oriented Interpreter Design 95
6.1. Windowing Environments 96
6.2. Memory Organization: Interpreter Objects 97
6.2.1. The Object Pool 98
6.2.2. Interpreter Objects 98
6.2.2.1. Frames 99
6.2.2.2. Binding Environments 101
6.2.3. The Interpreter 101
6.3. Interpretation 102
6.3.1. Unification 102
6.3.2. Clause Selection 105
6.3.2.1. Primitives 106
6.4. Summary 107

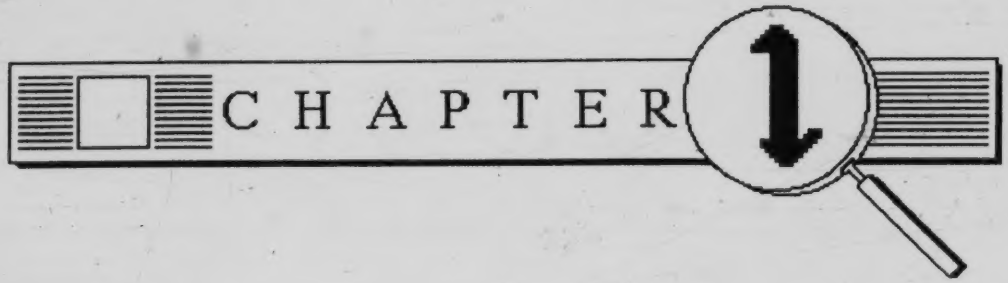
Chapter 7: Conclusion 108

Bibliography 114

Appendix 1: Horn Clause Solutions to Schubert's Steamroller 119

LIST OF FIGURES

Figure		Page
2.1	A component diagram of Godel	15
2.2	A Set addition predicate	21
2.3	Dialog window	28
2.4	An alert window	28
2.6	An IF template	29
2.6	Horn clause selection regions	30
2.7	A structure editor	31
2.8	If-then-else structure editing	33
2.9	An in-line dialog	35
2.10	A clause browser	37
2.11	A procedural debugger	41
3.1	Dialog editing of a function declaration	59
3.2	In-line dialog editing of a function declaration	59
4.1	Diagram of a Godel structure browser	74
4.2	Definition of the grandfather predicate	81
4.3	Class instances of a compiled Prolog procedure	82
5.1	Execution and Reset stacks with reference pointer orientation	91
6.1	Example unification methods for Godel parse nodes	104
6.2	Primitive clauses	107



Introduction

"What is the use of a book", thought Alice,
"without pictures or conversations?"

- Alice's Adventures in Wonderland, Chapter 1, Lewis Carroll

With the availability of high performance workstations and bit mapped graphical displays, programming environment designers are looking at a "gesture" oriented method of computer communication as an alternative to "natural language" approaches [Buxton 1987]. Though natural language understanding is an important topic in Artificial Intelligence research, the use of graphically oriented interactions as a "computer sign language" seems appealing.

In this thesis we wish to describe the marriage between a conventional batch oriented logic programming language (Prolog) and a user-interactive development environment. We believe the graphical interactive capabilities of single user workstations have not been fully utilized. Not only do the available multiple windowing systems create an effective environment in which to program, but the graphical manipulation and representation of code through syntax directed editors and module hierarchies can prove invaluable during the software development process.

This thesis also takes an in-depth look at the Prolog language and its interpretation mechanism. We will outline an architecture for a Prolog programming environment

designed for a graphically oriented single user workstation. We not only describe this architecture, but demonstrate how the environment can be used to add software engineering features like modularization and strong typing to the Prolog language. We also believe the debugging phase of Prolog program development can be improved through graphical animation of the refutation process. Both procedural and declarative debugging algorithms can benefit from the improved interface designs.

Our goal is not to create yet another Prolog interpreter, but to formulate new concepts in interactive graphical programming environments that complement or enhance existing Prolog systems. We describe each aspect of our environment including modularization, typing, open world programming, and debugging. We also argue that the object-oriented design methodology used in both the graphical interface and the inference engine provides a suitable foundation for the environment's architecture.

When designing a graphical interactive environment for Prolog, we kept the following in mind:

> Every good computer interface evolves over time, driven by vendors who see ways to stretch and extend the original idea. Sometimes those ideas work; sometimes they don't. But this isn't a holy war ...[1]

The resulting prototype Prolog environment was developed for just those reasons: to experiment with ways of making a more logical, appealing, and useable environment. We swayed from the compelling force to extend or modify the Prolog language, or increase the interpreter's execution speed. We took the approach that if Prolog is a language to be taken seriously, it must be complemented with a powerful and useable environment.

---

[1] Seymour, Jim, "Despoilers of The Interface?", MacUser, Vol. 4, No. 8, Aug. 1988, p. 77-78.

Our prototype was implemented using the Smalltalk-80[1] language [Goldberg and Robson 1983]. We chose this language for several reasons:

- "The Smalltalk-80 system is ideally suited to exploratory programming, software prototyping, simulation, modeling, AI research, and the *rapid development of sophisticated interactive applications*. Design and implementation of applications can occur simultaneously, encouraging experimentation and creative solutions to complex problems."[2]

- The availability of symbolic debugging , automatic garbage collection, and incremental compilation facilities in Smalltalk-80 will increase our prototyping productivity. This will allow quick testing of new interface and interpreter designs.

- The ability for future experimentation. Smalltalk-80 is a powerful language and has a powerful development environment. We wish to study the integration of Prolog within the object-oriented programming paradigm. By implementing our Prolog interpreter in Smalltalk, future versions can be extended to unify these two paradigms.

- The ability for the Prolog environment to utilize the vast amounts of code available in the Smalltalk-80 system by creating user-definable primitive clauses whose bodies are Smalltalk-80 source code.

1.1. Logic Programming: A Brief History

Historically, programming languages were developed as a bridge between the intricate workings of the machine and the abstract ideas of its users. When the feasibility of using computers to do larger and more complicated tasks became evident, a new means was needed to map increasingly complex algorithms into executable instructions. Programming languages tried to attain this by abstracting the computer's underlying architecture, permitting the developer to concentrate on what a program must do instead of how it would be done. Procedural languages like Fortran, Pascal and Ada[3] emerged, and proved useful in coding large software systems. Unfortunately, due to their complex syntactic and semantic constructs, they were difficult to manipulate in a

---

[1] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.
[2] Xerox Special Information Systems newsletter. Xerox is a trademark of Xerox Corporation.
[3] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

sound mathematical manner (such as proving program correctness). The programming language, Lisp, based on Church's lambda calculus, was one exception. It was created for its mathematical elegance instead of efficient execution on von Neumann architectures. This tendency towards elegance and simplicity instead of ad hoc concoctions of syntactic and semantic constructs is a vital part of programming language design. For this reason, programming paradigms based on mathematical logic have attracted considerable attention over the last decade.

Logicians and mathematicians, in their study of mechanical theorem proving, have opened a new area of programming language research by using techniques from mathematical logic to axiomatize a problem domain so a computer can "reason" about the problem it is trying to solve. This approach is far from being realized, but an important step forward was the invention of the resolution principle by Robinson [Robinson 1965] which formed the basis of many automated theorem provers. Further developments in automated theorem proving, using the resolution principle, laid the foundation for the invention of Prolog; an acronym for **Pro**gramming in **log**ic.

Prolog was first conceived by Alain Colmerauer and Robert Kowalski. It was based on a subset of first order logic known as Horn Clause Logic[1]. Prolog is a declarative language. The programmer describes known facts and relationships within the problem domain instead of describing the sequence of steps needed to find a solution. It was originally intended as a natural language processing system, but its usefulness was quickly established through the emergence of applications in areas of planning problems [Warren 1976], database query [Gallaire 1978], and natural language processing [Walker et al. 1987].

---

[1] Horn Clause Logic was first studied by A. Horn in 1951 [Ref]. Formulas in clausal form containing at most one positive literal are considered Horn clauses.

Development of logic programming interpreters and compilers has been under way for many years. While early logic programming interpreters (i.e. Prolog interpreters) were inefficient in both processing time and memory consumption when compared to conventional imperative languages, fast implementations were developed by being coded directly in assembly language. An example is Waterloo-PROLOG created by Roberts [Roberts 1977] at the University of Waterloo. Improved efficiency resulted by compiling Prolog procedures into an intermediate form and executing the compiled code. This was demonstrated by Warren's DEC-10 Prolog compiler [Warren 1980]. Additional work by Clark and McCabe [Clark et al. 1981], Mellish [Mellish 1982], and Bruynooghe [Bruynooghe 1982] have increased our understanding of prolog implementations.

Several Prolog implementations exist. Some are tailored for fast execution while others are experimental implementations designed to determine the feasibility of Prolog as a new programming language. Still others have been designed for experimenting with extensions to make a general "logic programming" language. To expand the scope of logic programming implementation research, a movement from coding in a machine specific language, such as assembly language, into higher level languages emerged. LM-Prolog [Kahn and Carlsson 1984], Waterloo Unix Prolog [Cheng 1984], NU-Prolog [Naish 1986] and others, written in Lisp and C, have proved portable across machines. This allowed widespread use of the language among a variety of computer systems. Moreover, these varying design goals have produced an abundance of Prolog interpreters.

Goals of portability and efficiency pose formidable problems in Prolog interpreter designs. Pioneering work by Warren in his design of the "W*arren Abstract Machine*" (WAM) [Warren 1983] proved that Prolog could be executed in time and space

equivalent to other symbolic languages. However, further efficiency gains are necessary for Prolog to become the implementation language of the Japanese initiated Fifth Generation Computing Systems (FGCS) project [ICOT 1982]. To increase efficiency, VLSI technology can be used to implement high level computer architectures (for discussions on the design of special purpose computers for logic programming see [Tick and Warren 1984] and [Mills 1986]).

Generally, it appears that sequential Prolog implementations have become as efficient as possible. Therefore, to increase performance further, a movement from sequential to parallel implementations is gaining momentum. These implementations take advantage of the inherit parallelism available in logic languages. Although such work is interesting, in this thesis we wish to deviate from the issue of fast logic inference engines and look in more detail at the software engineering and user interface issues.

1.2. Logic Programming Environments: The New Wave

In contrast to developing faster Prolog interpreters, a second area of research is associated with development environments and how they should support programming in Prolog. Development environments deal with three distinct problem areas: programming-in-the-small, programming-in-the-large, and programming-in-the-many [Muller et al. 1987]. Programming-in-the-small refers to the construction, analysis, compilation, execution, debugging, and testing of modules in a software system. Programming-in-the-large is concerned with aspects of specification, design, integration and maintenance of software systems. Programming-in-the-many is concerned with multiple programmers working on a single project. Issues such as access control, mutual exclusion capabilities, documentation, change logs, network access, and project management are also of importance. The general goal of

development environment research is increased productivity of designers, programmers, integraters, and maintainers to reduce costs of design, integration, and maintenance.

Increased computing power in the form of high performance workstations equipped with bit mapped displays and pointing devices has directed development environment research towards graphical user-interactive techniques. Sun[1] workstations and MacIntosh[2] microcomputers have shown the usefulness of graphical human-computer interfaces in increasing developer productivity. The result is the integration of programming languages and the computer's graphical environment exemplified in, among others, the Smalltalk-80 , Mesa [Sweet 1985], and Interlisp-D [Teitelman and Masinter 1981] systems.

As yet, very little work has been done in studying the application of development environments to logic programming. This thesis describes the results of a study of development support for some aspects of logic programming. Prolog's global name space and non-deterministic computational model suggest two major areas for exploration: organization of the clause database and program debugging. The need for special purpose Prolog development tools has long been recognized [Shapiro 1982], [Pereira 1986]. We try to extend these ideas by incorporating the features of modularization, typing, open world programming, and graphically oriented debugging into an integrated Prolog programming environment. The result is a prototype environment named **Godel**[3]: a Graphically Oriented Development Environment for Logic programming.

---

[1] Sun is a trademark of Sun Microsystems Inc.
[2] MacIntosh is a trademark of Apple Computer Corporation.
[3] Also, the surname of Kurt Gödel (born 1906), an Austrian mathematician who formulated the theorem (known as Gödel's theorem) that in logic and in mathematics there must be true formulas

Besides discussing features to make Prolog suitable for large software projects, this thesis describes the implementation of a Prolog development environment using the new and promising Object-Oriented Programming (OOP) paradigm. This paradigm is the basis of languages like Smalltalk-80, C++ [Stroustrup 1987] and Eiffel [Meyer et al. 1987], and requires a new way of thinking about the design and coding phases of the software development life cycle. Classing and inheritance are two features that make the Object-Oriented programming method a powerful tool for both prototype and production development of software systems. For this reason, a new architecture was developed to incorporate the complex control flow of Prolog programs into the object paradigm. Furthermore, the suitability of the object-oriented architecture for use in a graphical user-interactive programming environment for Prolog is described and analyzed.

1.3. Thesis Overview

Chapter two is an introduction to Prolog programming environments. It outlines some deficiencies of pure Prolog and argues for the software engineering features that Godel adds to the Prolog language. It also outlines Godel's graphical user-interface. We argue that the graphical representation of code and execution states can enhance the development of Prolog code. Chapter three is a detailed description of Godel's modularization and declaration system. It outlines the module semantics along with the syntax and semantics of the declaration system.

Chapter four is the first chapter describing Godel's implementation. We introduce the Guide programming environment and describe its object-oriented design. We then describe how this object-oriented architecture was instrumental in Godel's design and

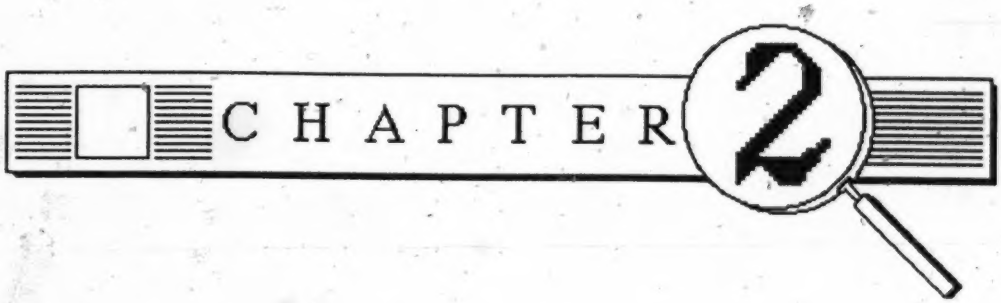
---

neither provable nor disprovable, making mathematics essentially incomplete, and the corollary that the consistency of such a system as arithmetic cannot be proven within that system.

implementation. We demonstrate how classing with inheritance enabled Godel to exploit Guide's architecture.

Chapter five is a general description of Prolog interpretation. It serves as an introduction to the unfamiliar reader and is used to describe important interpreter components. It also pinpoints areas for optimizations. Chapter six builds on this introduction by describing the object-oriented design of Godel's inference engine. It describes the interpreter's salient features and compares and contrasts it to a procedural design.

Finally, chapter seven concludes this discussion with a summarization of our work. It also outlines possible future enhancements.



Prolog Programming Environment Architecture

Good order is the foundation of all good things.

- Edmund Burke, *Reflections on the Revolution in France*

The concept of a ***Software Development Environment (SDE)*** has as many definitions as there are people to interpret it. We define an SDE as the collection of integrated tools used to design, code, and maintain a software system. Two important aspects of an SDE are the **integration** (communication) between tools and the scope of the tools. Here we are interested in the software coding level, where the set of tools are often termed a ***Programming Environment (PE).***

We consider tightly coupled (highly integrated) tools an important aspect of an environment's architecture. This chapter is a description of Godel's architecture, outlining the system's components and how we believe the subsystems must communicate to portray a unified view of development to the user.

2.1. Programming Environments

The architecture of a programming environment, independent of the language used, has three key components: a source code manager, a compiler/interpreter, and a debugger. Historically, these components were represented by a loosely connected web of independent tools with little communication between them. Usually, an editor

manipulated raw text while a compiler converted the raw text to executable form. If an error occurred, then the edit/compile cycle was repeated.

The evolution of programming environments from a chaotic pool of independent tools to a set of interacting components sharing a common intermediate program representation was a natural, evolutionary step. This efflorescence was stimulated by the falling costs and increased capacity of hardware, an increase in the complexity of programs, and a better understanding of software development from both theoretical and practical standpoints; all leading to richer, more robust environments.

The evolution of programming environment research has encompassed several major "classical" languages including imperative languages like Pascal and Ada and functional languages like APL and Lisp. The idea was to build an integrated environment that supported programming for a single language, provided a consistent view of the development process, and integrated the functionality of various tools used in the development process.

Regarding the evolution of the integration of software development environments, the best examples are the single language PE's of InterLisp-D and Smalltalk-80. The integration of tools in a single language PE is considered by many as a key to enhanced productivity and expressive power, and Logic Programming Environments (LPE) generally lack such integration.

Two approaches to unified LPEs are possible. The first approach is an LPE written entirely in a logic programming language. This is analogous to the InterLisp and Smalltalk PEs in which all environment functions are written in the language itself. Developing a programming environment for logic programming in a logic programming language has advantages and disadvantages. One advantage is the consistency gained

by having one development language. The editor, windowing system, module support, and other aspects of the PE could be conveniently and concisely expressed in a declarative manner. Extensions and modifications to the environment would be as easy and natural as regular logic programming development. A fundamental disadvantage is the newness of logic programming languages. Since they are young and evolving languages, much needs to be learned about their effectiveness for certain tasks. For example, there are no current standards for logic implementations of interactive interfaces, and it is not clear if their implementation is efficient enough with available logic programming technology.

The second approach is to implement the LPE in a second language. For example, the well established object-oriented design methodology and a corresponding object-oriented language could be used. Although there are drawbacks to this approach, there are several advantages:

- Object-oriented programming is designed for modular, reusable, modifiable code. During experimentation, the ability to make changes and enhancements to the existing source through class hierarchy changes is an asset.
- The object-oriented methodology is well suited for a graphical, window oriented application architecture [van der Meulen 1987], [Laursen and Atkinson 1987], [Grossman and Ege 1987]. This allows experimentation with the interaction between graphics and logic programming.
- It is possible that the object-oriented design of a logic programming inference engine may provide new insights into implementation strategies. In this thesis we will comment on the architecture of such a design and describe how it compares and contrasts to a standard procedural design.
- Object-oriented languages provide inheritance of class definitions. Similarly, taxonomic hierarchies have been used in knowledge representation. As in LOGIN [Ait-Kaci and Nasr 1986], this approach may provide an alternate architecture for combining inheritance into logic programming.

The disadvantages of this approach are:

- In the long term, LPEs should be written in logic programming languages. The object-oriented approach does not directly contribute to this goal.

- Object-oriented interpreter implementations may be slower than conventional procedural designs.

We believe the advantages of an object-oriented architecture outweigh the disadvantages at this time. It is for this reason that we implemented a prototype graphical logic programming environment (Godel) using an object-oriented programming language and environment (Smalltalk-80). It has borrowed features from existing procedural and functional environments, with the main emphasis being the following:

- It utilizes the hardware environment of a bit mapped, graphical windowing workstation;
- It makes the fundamental tool of the LPE an interactive editor;
- It provides editing commands that reflect the syntax of a program;
- It provides syntactic and static semantic checking;
- It reflects the name binding structure of programs;
- It provides source level debugging.

2.2. Prolog Environment Architecture

The architecture of existing Prolog interpreter environments consist of three independent components: a clause editor, an interpreter/compiler, and a clause database management system. This architecture is standard among all logic programming systems, but interaction between each has generally been kept to a minimum. For example, in the Unix[1] environment, the clause editor is one of the available editors such a Vi or Emacs. The interpreter is a completely separate software component that inputs objects from the editor, accepts user queries, and outputs results. During the debugging phase, if an error is found in a clause description, the developer must exit the interpreter, enter the editor to make changes, re-enter the interpreter to reload the clause definitions, and lastly to repeat the query and continue debugging. The number

[1] Unix is a trademark of AT&T.

of "task" switches is numerous, and any reduction would decrease development overhead.

To combat this, Prolog interpreters are usually equipped with three *system* predicates: edit, clearAll, and consult[1]. The clause edit(AFileName) invokes a task switch from interpreting to editing while clearAll and consult (or reconsult) are used to clear the internal database and reload the changed clauses. While friction between editing and debugging is decreased, problems arise because of inconsistencies created when reconsulting and loading files into the interpreter's workspace. If editing and debugging operations acted on a central clause database, the edit/debug process would be much simpler.

The decomposition of Prolog program development into clause management, interpretation, and debugging lends itself naturally to Godel's underlying architecture. We wish a highly integrated set of operators (tools) to act on a clause base so a uniform view of the system persists. Our design centers around the concept of a central clause repository (persistent heap) that is directly accessible by all environment components. The following is a list of the desirable features to include in a LPE. We do not claim that these are the only necessary features, but are the ones we have chosen to describe and implement:

- syntax directed editor
- modularization (clause management) system
- typing system
- graphically oriented procedural debugger
- open and closed world programming inference engine

[1] The actual spelling of these clauses varies from system to system, but the general idea remains the same.

The editor, inference engine, and debugging mechanism all have a consistent view of the clause repository. Similarly, through a graphical windowing system, the user has a consistent view of all the components within the environment. This is represented in figure 2.1.

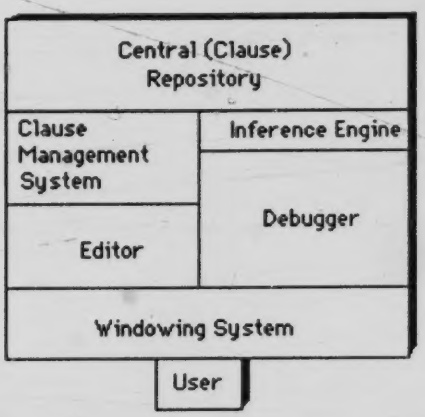


Figure 2.1: A component diagram of Godel.

The following is a description of Godel's clause management system, the editing subsystem, and the debugging subsystem, with emphasis on the inclusion of these features. Detailed descriptions of the clause database and inference engine are in separate chapters.

2.2.1. Central Repository

The central repository is similar to a clause database. Conventional Prolog systems maintained the clause database in a set of files, consulting each file as clauses are needed. Our approach is to view the clause database as one entity to aid in the development of a single program. The repository can be divided into modules which are shared among programs. The benefits of a central view of clauses is the ability for separate programs to re-use components.

The re-use of components among programs, and more broadly projects, is one advantage of a non-volatile repository for compiled clauses. The repository eliminates the need for re-compiling clauses by allowing them to exist indefinitely. This affects the design of the editor and inference engine since they not only operate on a clause base that is continually evolving, but they also manipulate compiled clauses instead of the clauses' textual counterparts.

Another advantage of the central repository is alleviating what we term the "dependency syndrome." PEs must be capable of answering queries of the form "Where are all the locations that <*anObject*> is used?" or "Where is <*anObject*> defined?." However, the time taken to locate declarations (definitions of variables, procedures, types (in imperative languages), functions (in functional languages), predicates (in logic languages)) increases as a software system becomes larger. The ability for the development environment to quickly answer these queries not only enables faster understanding of the dependencies that exist in a large software system, but also allows the developer to trace these dependencies.

2.2.2. Modularization

Very large problems are being programmed with procedural languages like Ada and Modula-2 through the use of modularization, an abstraction technique pioneered by Parnas [Parnas 1972a, 1972b] in which programs are decomposed into many independent modules which communicate using small, well-defined interfaces. The most common form of modularization is data abstraction in which a data type and all its operators are combined in one module. The implementation of these operations is hidden from users of the module and the operators form the users' interface.

The advantages of modularization generally, and data abstraction particularly, are well known and have been discussed by [Parnas 1972a, 1972b]. For example, modules provide scoping limitations so that names used locally in a module can be used independently in other modules. They limit the propagation of code changes since changes to the implementation of a module do not affect any other modules. They also provide portability and a good mechanism for isolating errors.

One method of introducing modularization to Prolog is through an LPE. An LPE can superimpose a module structure on the language without modifying the language itself. Introducing modularization through an LPE has three distinct advantages over creating a new language or modifying Prolog. They are:

- Since Prolog is widely used, there is a large body of existing code. Since code is often re-used, a new language would lose this library of existing code.
- The use of modularization in logic programming languages is a new concept and is not well understood. It will probably take some time before one modularization technique is recognized as superior. Thus, it is easier to experiment with an environment.
- There is no common syntax for Prolog, but a single environment can be configured to generate code for any dialect.

Existing module systems for Prolog are mainly concerned with the division of the predicate name space. The problem is that these systems are based on the syntactic view of modules (i.e. the compilers symbol table) instead of the view that modules have a connection with the language's semantic theory. Therefore, modularization should be geared towards constructing logic programs, and must diverge from the standard view used by languages like Ada or Modula-2. The approach we take is based on O'Keefe's algebraic formalization of a module system where he defines various operators used to construct programs out of pieces. His idea is that:

> existing module systems are based on a 'straightjacket' view of modules: that the function of a module system is to build

> very thick walls between modules and then let you chip tiny little holes in the walls. . . [O'Keefe 1985]

when in fact there should be varying degrees of module communication.

To define module communication in an LPE it is first necessary to understand the concept of clause extension. Godel's interactive environment relies heavily on incremental compilation. This affects not only the design of the clause database and inference engine, but also the interaction between a developer's actions and Prolog source code. We define incremental compilation in Prolog as the act of **extending** the clauses in a predicate or procedure. For example, if a procedure consists of the clause $p(X) \Leftarrow p(1) \wedge p(2)$ and the assertion p(1), then adding the assertion p(2) extends the procedure p to include this new clause.

In this thesis, two new modes of communication, or module operations, are introduced to complement O'Keefe's union operations. They are open inclusion and closed inclusion. Each of these operations represents a certain amount of glue between modules, with union creating the tightest binding and closed inclusion the least.

Union is defined as combining all clause definitions in one module with all definitions in a second module. With modules M_1 and M_2 represented as disk files, this is similar to the conventional Prolog environment where a user can consult(M_1) and then consult(M_2). Predicates declared in M_1 are indistinguishable from predicates in M_2. Naturally, any extensions made by the user after the union operation will add clauses to the existing procedures created during the union operation. As will be described in chapter three, union is a requirement for meta-programming and the all-solutions predicates.

The new **open inclusion** operation on modules M_1 and M_2 from within M_1 results in visible clauses in M_2 being brought into M_1. Any conflicts resulting from identical declaration names existing between modules must be resolved during the inclusion operation. As with the union operation, any clauses added to M_1 extend the definitions of M_2.

The last operation is **closed inclusion** and it has the least "side-effects" of the three gluing operations. Closed inclusion makes all public definitions in M2 available for use in M1, but M1 cannot extend the definitions in M2, nor can M2 see definitions in M1. This is similar to an import declaration in procedural languages where M1 wants to call procedure P in M2, but the implementation details of P are hidden.

The operations of union, closed and open inclusion do not affect the semantics of Prolog, but only constrain the visibility of predicates during program development (specific examples will be described in chapter three). For this reason, the gluing operations have a clear semantics, making their implementation straightforward. While other modularization systems for Prolog could possibly model these operations, it is Godel's interactive approach to error detection and correction that makes this scheme unique.

2.2.3. Typing

Many non-procedural languages like Smalltalk [Goldberg and Robson 1983], Lisp [Winston and Horn, 1984] and Prolog [Clocksin and Mellish, 1984] do not support compile-time type checking. Parameters to methods (Smalltalk), arguments to functions (Lisp) or terms in predicates (Prolog) not of the correct form are only detected at run-time. In large software systems, such argument mismatches may cause unexpected errors to occur in sections of code only executed in unusual situations, long

after product release. This is because complete path testing of large systems is impossible [Myers, 1979].

In the examples which follow, assume we have defined an operator, **add**, which adds an object to a Set. Further assume a logic error has resulted in the invocation of this operator in a situation where a constant is used as an argument instead of a Set.

In Smalltalk, argument mismatch errors lead to an object receiving a message that is not in its class's method dictionary. For example, the Smalltalk expression **$a add: 1** results in a notifier window which states that an instance of class Character ($a) does not understand the message **add:**. This results from the receiver of the message belonging to the wrong class.

In the situation where the argument to a message belongs to the wrong class, the error message is delayed until the implementation of a message causes another inappropriate message to be sent. For example, the expression **Set new addAll: 1** which tries to add all the elements from its argument collection to a new Set results in an error notifier indicating that an instance of the class **SmallInteger (1)** does not understand the message **do:**. This is because the message **addAll:** sends the message **do:** to its argument, 1, somewhere in its implementation and instances of class SmallInteger do not understand the message **do:**. These errors are not found at compile time as they would be in a strongly typed language, but they are detected and reported at run-time.

In Lisp, if the number of arguments does not match the number of parameters (impossible in Smalltalk or Prolog), it is reported when the function is invoked. If a parameter is mismatched, then the error can be reported when it causes an error in a built-in function (whether it is reported depends on the dialect of Lisp). For example,

(add 'a 1) could lead to an error when in the implementation of add, the interpreter checks if the atom 1 is in the atom 'a, which should have been a list.

In Prolog, the situation is worse. A run-time error will not appear since an argument mismatch will result in a unification failure and the interpreter will proceed. For example, **add(1, a, NewSet)** will simply fail when the terms **inList(1, a)** and **notInList(1, a)** in its definition fail to unify (see Figure 2.2 for a definition of add using notInList and inList and Figure 2.7 for definitions of notInList and inList). The user is never warned of the type mismatch.

```
/* add(AnObject, ASet, NewSet) is true if NewSet is ASet with AnObject   */
/* added to it.                                                          */
/* addToList(AnObject, ASet, NewSet) is defined in a List module.        */
/* For brevity, we have not included its definition.                     */
add(AnObject, ASet, NewSet) ⇐
        notInList(AnObject, ASet) ∧
        addToList(AnObject, ASet, NewSet).
add(AnObject, ASet, NewSet) ⇐
        inList(AnObject, ASet).
```

Figure 2.2: A Set addition predicate.

We are not claiming that Prolog is more susceptible to argument mismatch errors, only that it is more sensitive to them, since such errors are harder to detect. For this reason, it is essential to introduce some method of detecting argument mismatch errors.

The most straightforward solution is static type checking. This approach has been discussed by [Mycroft and O'Keefe 1984]. They proposed a parameterized polymorphic typing scheme which types the following Prolog objects:

- Predicates
- Functors
- Constants

For example, the predicate **add/3** can be defined as having three parameters of types: α, **List of** α, and **List of** α, where α is a parameter which has a type as a value. This way, we can use add/3 to insert a **Color** into a **List of Color**, and return the new **List of Color** or we can use add/3 to insert an **Animal** into a **List of Animal**, and return the new **List of Animal**. The predicates which use type parameters are said to have polymorphic types.

However, explicit variable type declarations are not necessary in an LPE since variable types can be inferred when the predicates which use those variables are entered into the environment. For example, given the predicate declaration **isEmpty(List)**, the environment can infer that the type of the argument variable, L, in the term **isEmpty(L)**, is **List**. This means that a user need only declare types for functors, predicates, and constants. We have used this approach for static type checking of Prolog in Godel.

When introducing types in a Prolog interpreter two questions arise: should typing be strictly enforced so that type declarations must be in place before declaring clauses, or should typing remain optional? In production software strong typing is a desirable feature, while in prototype development the abandonment of strong typing allows greater flexibility at the expense of security. Therefore, Godel supports the concept of a pseudo-strong typing system. This system allows a mixed state of well-typing to exist. Type information is specified interactively as the software is transformed from a prototype to a production form.

For example, a prototype project is developed without typing concerns. When the developer is convinced the design is appropriate, typing constraints are introduced. By using type declarations the types of predicates, constants, and functions are defined. Interactively, the developer can be notified of type inconsistencies within the code

allowing either reformulation of typing constraints, or reformulation of the otherwise ill-formed clause. We feel that a pseudo-strong typing system makes an LPE more useful over a greater range of applications, from prototypical work to production development.

Although other Prolog systems have static type checkers, the interactive ability to make a transition from untyped to well-typed source code alleviates the burden of constant mode switches required during the edit/compile loop of static type checkers.

2.2.4. Open World Logic Programming

[Komorowski and Omori 1985] and [Sergot 1983] have described a programming environment in which the user is prompted for more information if unification fails. Specifically, if the environment is trying to unify an open predicate, P, and there is no clause whose head will unify with P, then the user is given the choice of adding more information about P in the form of new clauses whose heads are P. Also, they have provided a mechanism by which any existing predicate may be explicitly opened or closed.

On one hand, after the definition of the predicate includes(List, Object) which checks to see if an object is in a list, the predicate should be closed so that the environment does not query the user when the end of the list is reached. On the other hand, consider the situation in which the user enters the query grandmother(Who, John). Suppose that during unification, the environment generates the goal mother(X, John) and cannot unify this goal. If the predicate mother was open, then the environment could query the user for more information. If the user enters a clause which identifies the mother of John, then unification would proceed.

The ability to add new information when unification fails is called the open world assumption and is not commonly included in logic programming languages. The open world assumption has two practical advantages in logic programming. The first is that it allows an application to operate with incomplete knowledge and provides a simple mechanism for the application to gain knowledge from a user. The second is that it is a good interactive technique for developing applications since the programmer is prompted for missing clauses as their necessity is determined.

The open world assumption interacts with modularization in an interesting way. Since the implementation of modules is private, it is not appropriate for the user to supply new clauses to a module, when using the module. This means that a programming environment which adds both modularization and the open world assumption must be able to close all predicates in the module.

In Godel, there are two possibilities for using the open world programming method. The first is when a goal (with associated open predicate) fails. Each predicate has an attribute that specifies if it is open (extendable), or closed (non-extendable). In the case of an open predicate, Godel creates a debugger window on the failed predicate. This indicates to the user that more information may be entered to satisfy that predicate (or execution can be merely continued).

The second possibility is when no corresponding clause exists for a predicate declaration. Although Godel's static semantic checking alleviates many possibilities for using an undeclared predicate, it does not address the issue of an undefined predicate. If a predicate has no corresponding clause, a debugging window is opened on the current refutation, highlighting the undefined predicate. The user can then use the code browser to enter more information and continue execution.

2.3. Overview of Godel's User Interface

We have described features added to Prolog to make it suitable for large software development. Our goal, though, is the integration of these feature within a coherent environment running on a single-user workstation that includes a high resolution raster display and a pointing device[1]. The following section is an overview of Godel's user interface, describing the interaction paradigm between user and environment.

2.3.1. User Interface Paradigm

The investigation of the effects of graphical user interfaces on programming environment design has led to new ideas about the use of graphics for representing, editing, and constructing programs. The graphical representation of programs can influence language directed editing, approaches to incremental compilation, and debugging techniques. We have extended these ideas from the procedural programming domain and will show their usefulness in a logic programming environment. Many ideas developed here originate from environments like PECAN [Reiss 1984], Magpie [Delisle 1984], and GUIDE [Szafron and Wilkerson 1986a].

Godel's interface was designed with three concepts in mind. The first is that a consistent, single interface model should be available to the user. The ability to move between components without mental context switches or remembering component dependent commands is seen as vital in designing an effective user interface. The second concept is how to handle incomplete and incorrect programs. We take the approach described in [Szafron and Wilkerson 1986b] in which the environment is always kept semantically correct by using dialog windows that house information that

[1] Most workstation pointing devices are called a "mouse." We will use this term throughout.

is either waiting initial entry into the system or is semantically incorrect. The last concept is that a graphical interface may affect the run-time environment. We wish to use graphical techniques to enhance the debugging of logic programs. The following is a description of how the interface is designed to produce these effects.

2.3.2. Interface Components

The developer interacts with Godel using a pointing device and a set of windows. Windows contain rectangular panes representing views onto aspects of the environment. These views include clause definitions, type definitions, module structures, and run-time traces. Windows can overlap each other, so navigation or context changes occur by simply moving the mouse from one window to another. The active window is a window with special status since no other window may overlap it. It is denoted by a highlighted title bar. A window is made active by using the mouse to point to it and pressing the mouse button. That same button selects character positions or structures within a pane.

Commands are entered using pop-up menus. Each pane in a window has a menu associated with it. The menus that appear are context sensitive in that their contents are partially determined by the current state of the pane and the types of structures visible or selected.

A structure in a pane is selected by moving the mouse pointer over the structure and pressing the mouse button. Selected structures are denoted by highlighting them with a contrasting background. Several structures may be selected by dragging the mouse while the button is pressed. Special actions may be performed on a structure by double

clicking[1] the mouse button while over the structure. The action taken depends on the current window and the selected structure, but usually an edit (dialog) box is opened on that structure.

2.3.2.1. Dialogs and Alerts

Dialogs are a class of windows used to convey or retrieve specific information, and are classified into two categories: modal and modeless [Inside Macintosh 1985]. Modal dialogs (also called alerts) do not allow the user to switch contexts, but enforce explicit acknowledgement of the window's contents (message). In contrast, modeless windows allow the user to switch between contexts without being constrained to one "mode" of operation. Modal dialogs usually contain error information, or ask for important information that is required before execution can continue. On the other hand, modeless dialogs contain information that does not have to been immediately entered into the system (i.e. information that is not required by another object).

In Godel, dialogs hold textual information that has yet to be accepted into the system or that was excised because of semantic inconsistencies. A dialog has a title tab that displays the module name and the type of declaration, plus window panes containing text and associated labels indicating the text's type (see figure 2.3). By encapsulating semantically incorrect objects in dialogs, the clause base can always be kept semantically consistent. The main use of modal dialogs in Godel are as alert boxes (see figure 2.4) that convey error messages to the user.

---

[1] Double clicking is the action of pressing the mouse button twice within a specified time period. This time period is usually about 250 ms, but varies between implementations.

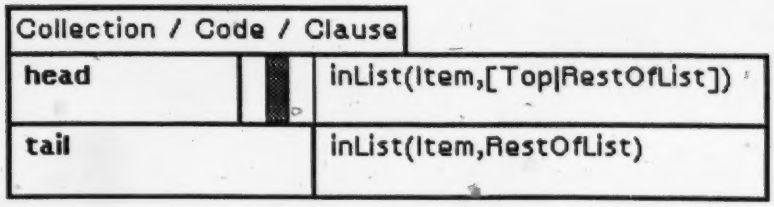


Figure 2.3: A dialog window for entry of source code (a clause) in the module Collection.

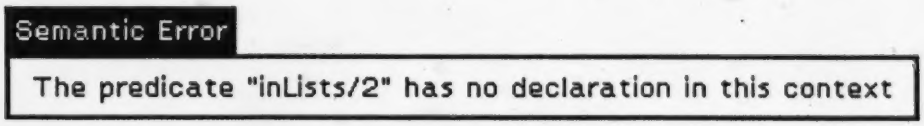


Figure 2.4: An alert window for undeclared predicate inLists (instead of inList).

2.3.2.2. Language Directed Editing

The divergence from conventional text editors to editors that reflect the syntactic and semantic nature of programming languages provides several advantages in the programming environment domain. A syntax directed editor, as described in [Tietelbaum and Reps 1981], produces language templates that alleviate the need for the user to remember specific syntactic details. Therefore, syntactic errors can be eliminated, or detected during code entry rather than at compile time.

The design philosophy of Godel's editor is to detect and correct errors as early in the development cycle as possible. The editor compiles all new input and performs both syntactic and static semantic analysis on this input. This method differs from the batch oriented compilation process of conventional interpreters in one major respect: Godel's incremental approach. Our editor provides user feedback as changes are made to clauses, without having to recompile the entire module (or file) where those clauses reside. It is our belief that this early error detection can increase developer productivity.

One problem associated with the design of syntax directed editors is the scoping level of syntactic constructs. That is, how many syntactic levels of the language should the user be required to enter. An example is the IF statement of procedural languages (shown in figure 2.5). Entering an editor command (via control keys, menus, or other input techniques) creates a template for an IF statement. Now, there are two templates needing more information: the condition and the body. The user may use syntax directed commands to create the body of the IF, but it is too cumbersome to use syntax directed editing on the condition of the IF. In such places, it is desirable to enter a "text editing" mode where source code can be entered directly. For example, the condition "x < 10" could be typed directly into the IF condition template instead of generating further templates for the two expressions (the variable x and the constant 10) and the operator "<".

Figure 2.5: An IF template: The outlines show the selectable components of the template. The <condition> is at a scope where conventional text editing is suitable.

The problem of distinguishing between using syntax directed editing and conventional text editing is more pronounced in Prolog than some other conventional languages because of Prolog's use of Horn clauses and their simple syntax. The approach we take is a combination of straight textual editing and syntax directed editing. Since there are a few syntactic constructs, syntax editing of atoms within a clause would produce too much overhead. On the other hand, the ability to select clauses or parts of clauses has a positive point: modules can be kept semantically correct at all times by performing semantic checks on a "copied" or "cut" structure before "pasting" it into the new module.

The structure editing of clauses uses the following formula. For simplicity, we define "selecting an object" to mean moving the pointing device to that object and pressing the mouse's selection button. To select a literal in a clause, select any part of that literal. To select the entire body of a clause, select a literal in the body and drag the mouse while depressing the mouse selection button. Alternately, select a conjunction sign between any two literals in the body. To select an entire clause, select the clause's implication symbol. To select several clauses, select a single clause and, while pressing the mouse selection button, drag the mouse over those clauses. These selection regions are shown in figure 2.6.

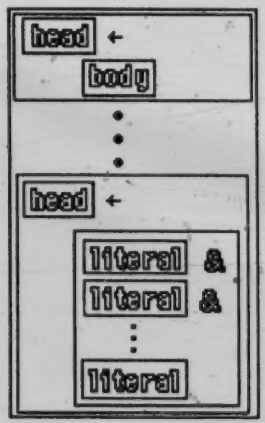


Figure 2.6: Horn clause selection regions : the outlines show the selectable structures.

When entering clauses textually, the user types directly into a dialog window (or a text window in the case of in-line dialogs described in section 2.3.2.4., **In-line Dialogs**). Accepting the input allows the editor to make immediate syntactic and static semantic checks, reporting errors to the user using an alert window. If no errors occur, the parse tree for the clause is created and a printable representation of that tree is displayed in the edit window (see figure 2.7). From then on, the user can select and manipulate (using the workstation's pointing device) components of a clause.

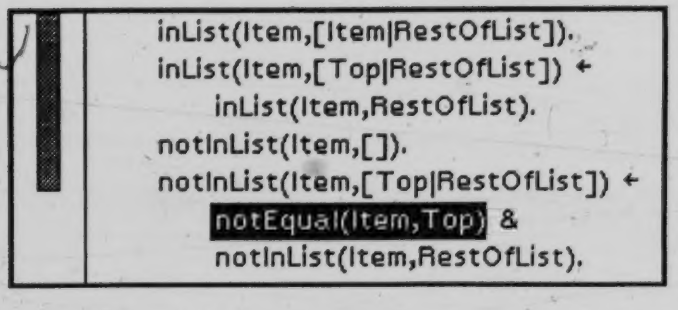


Figure 2.7: A structure editor: selecting an atom within a clause

2.3.2.3. Graphical Clausal Representations

Language directed editing fits naturally into an interpreted language like Prolog because all program clauses must be compiled into an intermediate form for fast execution. By editing the compiled forms of clauses instead of their textual representation, incremental compilation of Prolog programs is straightforward. Not only does incremental compilation benefit from syntax directed editing, but so does the graphical representation of Prolog programs.

Since the editor knows about (and can manipulate) the syntax of clauses, it can automatically format source by making its own syntactic transformations. Note however that we are not claiming our scheme for clausal representation is more readable than another scheme, for this is a matter of taste. What we are trying to demonstrate is that the editor can be used to alleviate the burden of formatting source code, simultaneously allowing the addition of special syntactic constructs to the language.

A prime example of this is the if-then-else construct incorporated into most Prolog languages. If-then-else is used as a soft cut, eliminating the the explicit use of the cut/0 predicate by the programmer. If-then-else is sometimes implemented directly into the

inference engine, or can be defined using cut/0 and prove/1 (see section 3.2.1.2., **The Meta-predicate Prove**) as:

```
ifThenElse(X, Y, Z) ⇐                    if X then Y else Z ⇐
        prove(X) ∧                               prove(X) ∧
        cut ∧                                    cut ∧
        prove(Y).                                prove(Y)
ifThenElse(X, Y, Z) ⇐                    if X then Y else Z ⇐
        prove(Z).                                prove(Z).
```

Historically, Prologs used a syntax like (<cond> -> <thenBody> ; <elseBody>) to represent the if-then-else. An example of where an if-then-else would be used is in a deterministic membership predicate which is true if its first argument is a member of the collection represented by its second argument. The implementor assumes the first argument is always ground and wants no backtrack points to remain once membership is satisfied (for efficiency sake). This behavior can be implemented in at least three ways:

```
/* conventional definition for member */
member(Item, [Item | RestOfList]) ⇐ cut.
member(Item, [FirstItem | RestOfList) ⇐
        notEqual(Item, FirstItem) ∧
        member(Item, RestOfList).

/* conventional (i.e. CProlog) syntax for member using if-then-else */
member(Item, [FirstItem | RestOfList]) ⇐
        ( Item = FirstItem ->
                true ;
                member(Item, RestOfList) ).

/* Godel syntax */
member(Item, [FirstItem | RestOfList]) ⇐
        IF Item = FirstItem THEN
                true
        ELSE
                member(Item, RestOfList)
        ENDIF
```

We claim that the last version is more readable, and fits naturally into the language directed editing paradigm that Godel supports. The editor stores the if-then-else in clausal form which can be proved via the defining if-then-else clauses:

```
member(Item, [FirstItem | RestOfList]) ⇐
        ifThenElse(=(Item, [FirstItem|RestOfList]),
                        true,
                        member(Item, RestOfList)).
```

However, it is displayed for editing in its expanded form. This idea can be extended to constructs such a repeat/fail loops. See figure 2.8 for an example.

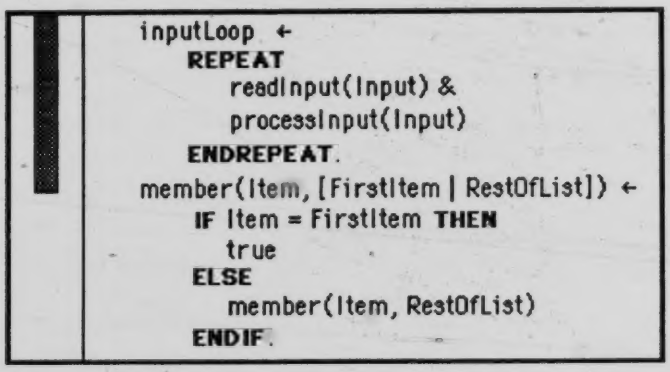


Figure 2.8: If-then-else structure editing (along with a repeat construct).

The debate on the use of if-then-else and other procedural notations has raised some concerns over their usefulness, or for that matter, their appropriateness. As Shapiro pointed out [Shapiro 1982], the use of if-then-else is geared towards an operational rather than declarative reading of clauses, results in writing complex clauses, and needs explicit equality calls to instantiate output variables. For these reasons, our ideas of using syntax directed editing of these constructs should only be interpreted as follows: the use of procedural constructs are neither beneficial, hindering, or mandatory. Only that, if they are used, they should be treated procedurally.

2.3.2.4. In-line Dialogs

The goal of Godel's language directed editor is to keep the clause database semantically consistent at all times. When syntactic and semantic errors occur during the incremental compilation of Prolog clauses, they cannot be acknowledged for later modification, but must be refused acceptance into the system. This means that the user must re-submit the incorrect information after taking corrective measures. Where is this information stored in the interim?

The solution we use is a variant described in [Szafron and Wilkerson 1986b] (and implemented in GUIDE) whereby the user is required to enter code and data through special dialog windows. As correct information is entered it is transferred from the dialog to another window holding a correct program version (called the program clause window). This scheme is useful in solving the previous problem of dealing with incorrect programs, although we have modified it slightly to allow a combination of conventional text editing and dialog entry.

We introduce a third type of dialog called an **in-line** dialog. Instead of creating a separate text entry window, the in-line dialog is embedded within the program clause window (see figure 2.9). The user initially selects (via a pointing device) the position he wants text entered; we call this position the *insertion point*. If the insertion point lies between existing structures, then typing will insert characters between those structures. The in-line dialog is represented by a thin lined border surrounding the inserted characters. This not only gives the user the visual clue that new text is being entered but also outlines the text's extent. During this phase, the user is in text entry mode where the selection of structures outside the bordered text is not permitted, although within this border conventional text editing is possible. If accepting the text does not

lead to an error, the in-line dialog disappears and the text is automatically reformatted. If errors do occur, the user can make changes and try re-accepting, or excise the text into a conventional dialog to escape the in-line dialog mode.

The advantage of this approach is the ability to use the dialog entry paradigm while retaining the more familiar text entry paradigm of conventional editors. Also, the context switch from editing the program clause window to editing in a dialog is alleviated, allowing more editing to be done in one window.

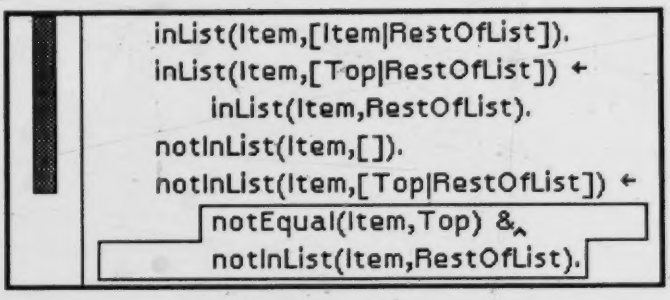


Figure 2.9: An in-line dialog: the border indicates the in-line dialog start and stop. The insertion carat indicates where typed text is inserted.

Syntactic and semantic errors are important aspects of program entry. In Godel, whether in-line or conventional dialogs are used, an error occurring during the acceptance of text produces either an alert dialog containing a message describing the type of error, or an error message inserted into the input text at the position where the error occurred. When an alert dialog is used, the text in the input dialog causing this error is highlighted to give a visual indication of the error's position.

2.3.2.5. Code Browsing

Godel provides access to program clauses and execution states through windows called browsers. Browsers are used to navigate through the module hierarchy or the refutation tree to access and modify program clauses. The **code browser** contains panes

to view: modules, the categories of modules, clauses and declarations within a module, and the gluing constructs between modules. The **debugger** is a special browser that displays the computation state of a query and is the interface to Godel's debugging algorithms. The last browser, the **workspace**, displays the history of Prolog queries and their variable bindings. Each module owns an instance of a workspace, where the code of the module can be executed.

The code browser contains six panes. The top left pane contains a list of categories, where each category has an associated list of modules. Each module in the system is assigned a category so it can be located more efficiently. The module pane contains a list of modules owned by the currently selected module category. By selecting a module within the module pane, the remaining five panes update their views to display information particular to that selected module. Two panes display information associated with module gluing operations. The first pane contains a list of modules used by the selected module either by union or inclusion operations. The second pane displays a list of clauses and declarations visible to external modules.

The other three panes are associated with program clauses for the selected module. They are the declaration protocol pane, the declaration pane, and the clause pane. The declaration protocol pane is similar to the module category pane. It is used to classify declarations. This allows efficient access to particular declarations. The declaration pane lists all declarations for the selected module while the clause pane contains all of the clause definitions for the module. The general design of the clause browser is to encapsulate in one window all information necessary to find and update program source efficiently (see figure 2.10).

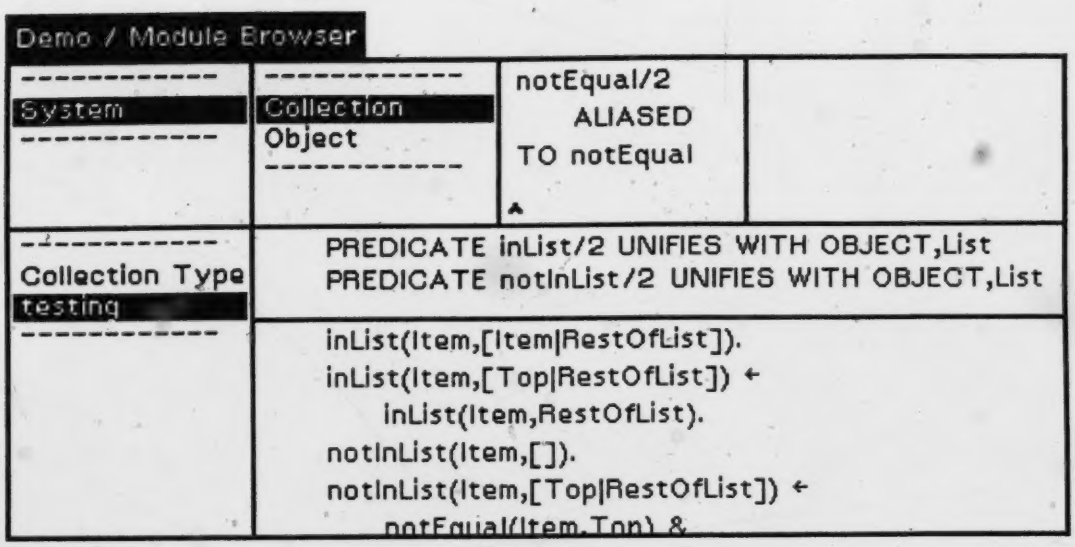


Figure 2.10: A clause browser: clockwise from the top left - module category pane, module pane, import pane, export pane, declaration pane, clause pane, declaration protocol pane.

The final browser, the workspace, is a conventional text edit window. Commands are entered, executed, and the results from the execution are displayed within this window. This replaces the usual screen prompt mode of conventional interpreters.

2.3.2.6. Procedural Debugging

There are two separate approaches to debugging Prolog programs: procedural and declarative (or algorithmic). The procedural (or trace) method closely follows the execution order of the inference engine. In contrast, the declarative (or diagnostic) method searches through a program's computation tree and asks the user to verify predicates based on their currently instantiated variables. From this information, erroneous program clauses can be uncovered. Two important works concerning the declarative debugging formalism are [Shapiro 1982] and [Pereira 1986].

In Godel, we have chosen to design and implement a graphical procedural debugger rather than a declarative debugger for three reasons. First, the procedural method is easy to incorporate into the interpreter. This allows quick prototyping to determine the feasibility of our ideas. Second, it is not clear if the declarative form of debugging is suitable for large programs since the debugging process is controlled by the system. Third, because of the complexity of declarative debugging algorithms, it may prove advantageous to implement them as meta-interpreters in Prolog itself. Future extensions to Godel could experiment with incorporating declarative algorithms into the inference engine. This would provide a comparative basis for the two approaches.

2.3.2.6.1. Debugging Models

There are two commonly used models for procedural debugging: the tree model and the box model. These two models are designed so a programmer can easily trace and understand the complex execution pattern of his program. The tree model relies on the refutation tree formed by the depth first left to right computation of pure Prolog . From this picture it is easy to see how a goal is divided into subgoals.

In contrast, the box model represents the flow of control through each subgoal. This model is simple, elegant, easy to implement, and is used by numerous Prolog implementations. It consists of four components, or corners of the box: call, fail, redo, and exit. **Call** is when a goal is first tried. **Fail** is when the goal's subtree fails to succeed. **Redo** is when a goal is repeated (a backtrack point) because other candidates exist. **Exit** is when a goal's subtree succeeds. The main drawback is that most box model debuggers are designed for a line-oriented display device. Godel, on the other hand, is designed for a bit-mapped graphical display. For this reason, we have transformed the box model to utilize the workstation's graphical capabilities.

Godel's procedural debugger consists of four components: a run-time stack pane, a code pane, a variable list pane, and execution buttons. The run-time stack pane represents the goals and their associated subgoals produced during a refutation. This pane is the history of the current refutation. By selecting elements from this list, the user can peek into the refutation's history to view clauses and their variable bindings.

The code pane is the link between the interpreter's stack frames and the user's source code. It contains actual source code instead of some intermediate representation of the program. If the debugger can highlight the currently selected goal, it is easier for the user to make the connection between the source and a point in the refutation tree.

The variable list is the link between the source code's actual variable names and their bindings. This list consists of variables *as they appear* in the user's source. This eliminates the need for cryptic variable identifiers like _123 when browsing variable bindings. Internal variable identifiers are still required when viewing variables bound to structures who in turn have unbound variable components. For example, suppose the list structure [Head | Tail] has its two variables unbound. Suppose also that the variables X and Y are bound to this structure. When browsing variable bindings within the debugger, we need a visual clue that both X and Y are bound to the same list. In Godel, we superscript each variable with a unique number to identify unbound variables. We also print the variable name in a contrasting font. For example, X = $[Head^{1234} \mid Tail^{1235}]$[1].

Godel's debugging model is a variant of the box model. We wish to display the four actions of *call*, *exit*, *fail*, and *redo*. To do this, the debugger provides three buttons.

---

[1] This output representation has not yet been implemented.

Commands are sent to the debugger either through pop-up menus or these buttons. The execution buttons, though, allow for faster command access. The *step* button is used to proceed one unification at a time. It allows the user to enter into a subtree. The second button, *call*, is used to execute an entire goal, skipping the subtree rooted at that goal. The last button is the *fail* button. It warns the user that the current operation failed. From a failed node, the user can use the step button to step into the call and try to find the section of that subtree that caused the failure, or he can press the *redo* button (located below the fail message) to advance to the last backtrack point. An additional button, *proceed*, is used to continue until a breakpoint is reached or the refutation completes.

2.3.2.6.2. Breakpoints

In a trace and spy Prolog debugger, it is necessary to set breakpoints in a variety of locations. The user may wish to halt execution whenever a successful unification occurs with any clause head within a procedure. The user may also wish to set breakpoints between or directly on any atom within a particular clause. Inserting breakpoints between atoms is analogous to adding a primitive system predicate breakPoint/0 whose body notifies the interpreter that a breakpoint was reached. This method, though, requires the user to assume there is a procedural and predefined ordering of goals within the body of a clause, an unrealistic assumption in a true logic programming language.

We argue that breakpoints set on atoms must specify one of two conditions: break before attempting to solve the goal, or after the goal is solved (denoted *breakBefore* and *breakAfter*). Godel will stop execution and display the current state of execution in the debug window when encountering a breakpoint. Breakpoints are set by selecting an

atom within a clause and selecting the "add breakpoint" item from the clause menu. Clauses with breakpoints are displayed with a different font to give a visual clue of their presence. For example, in the clause grandfather(X, Y) ⇐ father(X, Y) ∧ father(Y, Z), if the user sets a *breakBefore* on father(X, Y), he can examine the bindings for X and Y, but Y will still be unbound. If he sets a breakAfter on father(X, Y), he can examine Y's binding.

2.3.2.6.3. Execution Visualization

Figure 2.11 is an example debug window that provides a visual representation of the execution state. The query **notInList(5, [1, 2, 3, 4])** was issued. We have stepped to the point where a comparison (notEqual) is being performed.

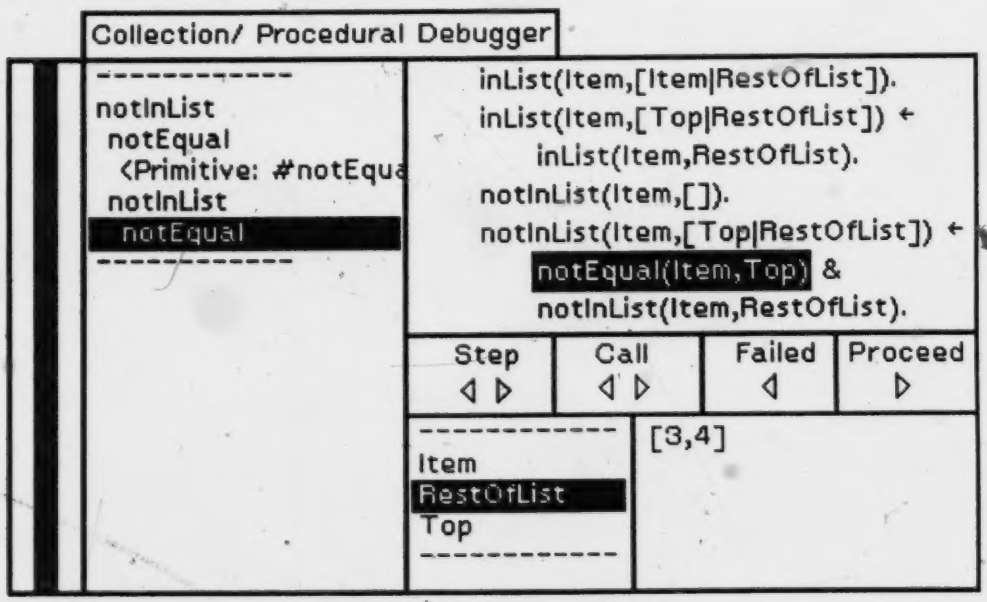


Figure 2.11: A procedural debugger

At this point, if the user selects the right pointer under the call command, the highlight will move from the atom **notEqual(Item, Top)** to **notInList(Item, RestOfList)**. Alternately, if the user selects the right pointer under the step command, the source

code pane changes to display the defining clause for notEqual. When the recursive goal notInList succeeds, an insertion carat is placed at the end of the clause. If a step or call is activated, execution returns to the clause's parent goal. In the original queries case, this success exit results in all variable bindings being displayed in the debugger's associated workspace window. The left arrow under step and call is used to step backwards (this has not yet been implemented).

2.4. Summary

In this chapter we have described the components we believe are needed for large Prolog program development: modularization, typing, and open world programming. Similarly, we have outlined the general architecture that our environment must possess, the main component being the central clause repository. This repository can be considered a global database (or knowledge base) that is used to construct modular programs.

Lastly, we introduced the concepts of Godel's user interface. We also showed how the interface manipulated the software engineering components described earlier. We introduced features that rely not only on the graphical nature of the environment, but on the environment's underlying architecture. Since the editor manipulates a constantly evolving clause base, features like language directed editing and incremental compilation fit naturally into Godel's overall design.

Prolog Development Environment Design Details

You can't invent a design: You recognize it, in the fourth dimension. That is, with your blood and your bones, as well as with your eyes.

- David Herbert Lawrence 1885-1930

From the previous chapter, the reader should have a strong grasp of Godel's architecture from a user (user-interface) point of view. In contrast, this chapter describes Godel's design from an implementor's point of view. It outlines the semantics of the modularization system, the syntax and semantics of the type inference system, and lastly briefly introduces the syntactic and static semantic errors detectable during incremental compilation of types and clauses.

3.1. Preliminary Definitions

The following section is a brief background on Herbrand Interpretations since they are used to explain Godel's module semantics. Also, an introduction to the BNF notation used in defining Godel's type and clause declaration syntax is described.

3.1.1 Herbrand Interpretations

A first order language L is a set of all formulas built from the alphabet having the following classes of symbols:

V: variables
C: constants
F: functions
P: predicates
C: connectives: $\wedge$, $\vee$, and $\Leftarrow$.
Q: Quantifiers: $\exists$, $\forall$

We assume that from this alphabet the usual definitions for terms, atomic literals, and well formed formulas are drawn. For Prolog, a program is a database of well formed clauses. Each clause is an expression of the form $H \Leftarrow G_1 \wedge G_2 \wedge ... \wedge G_n$ where H is considered the head of the clause and $G_1 \wedge ... \wedge G_n$ form the body of the clause and each G_i is considered a goal. A unit clause (assertion) is a fact in the database and has no goal in its body. A rule clause has at least one goal in its body. A clause's head and goals are all atomic formulas (or atoms) of the form $p(t_1, ..., t_n)$ where p is an n-ary predicate and $t_1, ..., t_n$ are variables or terms. We use the notation p/n to denote a predicate of arity n. Terms are either constants, variables, or functions of the form $f(t_1, ..., t_n)$ where f is an n-ary functor and $t_1, ..., t_n$ are terms. The actual syntax of a clause varies from implementation to implementation, so in this thesis we will use the symbol '$\Leftarrow$' to denote logical implication and the symbol '$\wedge$' to represent logical and.

The **Herbrand Universe**, U_L, is the set of all ground terms in L. The **Herbrand Base**, B_L, is the set of all ground atoms in L. A Herbrand **Interpretation** is a subset of B_L, with a ground atom A true in an interpretation I if and only if A is an element of I. Also, given an interpretation I and a first order language L plus a closed formula F, then I is a **model** for F if the truth value of F is true in I.

In the theory of logic programming, models and Herbrand Interpretations are used to define the concept of a least Herbrand Model (denoted M_P) such that:

$$M_P = \{A \in B_P: A \text{ is a logical consequence}^1 \text{ of program P}\}$$

The least Herbrand model is the set of elements in the Herbrand base that are a logical consequence of a program P. M_P defines the meaning of a logic program P, but to precisely define this meaning we must first define a mapping, T_P, from interpretation to interpretation such that:

$$T_P(I) = \{ A \in B_P: A \Leftarrow A_1 \wedge \ldots \wedge A_n \text{ is a ground instance in P and } I \supseteq \{A_1, A_2, \ldots, A_n\} \}$$

This represents a single deductive step for a logic inference engine. Therefore, to define the meaning of a program in terms of T_P, we must find the least fixpoint of T_P (which incidentally is M_P):

$$M_P = \text{lfp}(T_P) = T_P\uparrow\omega(\{\})^2$$

where

$$T_P\uparrow 0(I) = I$$
$$T_P\uparrow n(I) = T_P(T_P\uparrow n\text{-}1(I))$$
$$T_P\uparrow\omega(I) = \cup_{n<\omega} T_P\uparrow n(I)$$

This result is from [van Emden and Kowalski 1976] and can be found in [Lloyd 1984]. It represents all the grounds facts that can be obtained from program P in a finite number of steps.

3.1.2 BNF Notation

The production syntax for type declarations in this chapter follows closely the notation for extended BNF except terminal symbols are uppercase identifiers or surrounded by single quotation marks. As with extended BNF, optional categories are enclosed

---

[1] Let S be a set of closed formulas and F be a closed formula of a first order language L. We say F is a *logical consequence* of S if for every interpretation I of L, I is a model for S implies I is a model for F [Lloyd 1984, p. 14].

[2] Note that ω is the first limit ordinal other than 0.

within brackets ('[' and ']') and alternation (or) is represented by a vertical bar ('|'). We use the symbol * to denote the closure operator which represents one or more occurrences of an expression.

3.2. Modularization in Logic Programming

The standard interpretation for the meaning of a logic program is the least fixed point of T_P which views a program P as a set of facts that logically follow from P. When a program P is divided into smaller components (modules), it does not suffice to view the meaning of each of the modules as a set of facts, but as a mapping that takes facts and deduces new facts from them. In this way, we can define the meaning of a set of modules by defining their interactions. The interpretation we take is the same as in [O'Keefe 1985] and is such that a module is a mapping from interpretations to interpretations defined by:

$$M[\,[P]\,](I) = T_P\uparrow\omega(I) \text{ where P is now a program fragment (module)}$$

This notation assumes that M[[P]] is no longer a set of facts, but a function that takes a set of facts, I, and deduces new facts.

Modules for a programming language are used not only to create separate name spaces for symbols in the language, but to provide an organized means of constructing large programs out of smaller pieces. In the context of logic programming, this construction takes on a different view since modules contain axioms and axioms describe what is true or false in a world. By "gluing" axioms (modules) together, the meaning of the new module may not coincide with the meaning of the two independent modules. i.e. it might be that:

$$M[\,[M_1 \cup M_2]\,](I) \neq M[\,[M_1]\,](I) \cup M[\,[M_2]\,](I)$$

For example, if module M_1 contains axioms $p \Leftarrow q$ and $q \Leftarrow r$ while module M_2 contains the fact r, then p is false in $M[\,[M_1]\,](I) \cup M[\,[M_2]\,](I)$. Taking the meaning of M_1 and M_2 separately, p cannot be proven because r is not visible within M1, but taking $M' = M_1 \cup M_2$ then p is true in $M[\,[M']\,](I)$. It is the construction of a logic program out of fragments that can lead to the unexpected behavior of a program. By having a well defined interface between modules, the programmer is able to have better control over the interpretation of the program clauses.

3.2.1 Module Glue

We wish this well defined interface to have a semantic definition similar to the semantic definition for logic programming. To do this, we will define the functions (operations) that can be performed on modules when gluing them together. Since modules are a mapping from interpretation to interpretation, it makes sense to compose two modules together where the facts generated by one module are used by a second module. We define this operator as function composition and denote it by $\circ$ (first introduced in [O'Keefe 1985]):

$$M[\,[M_1] \circ [M_2]\,] =_{def} M[\,[M_1]\,] \circ M[\,[M_2]\,]$$

This is the situation where all the definitions in M_2 are available to M_1, but not in reverse. We also define the function rename(P_1, ..., P_n) to be a mapping that renames its arguments so no name conflicts appear. Thus, the composition of rename(predicates(S)) $\circ$ M results in renaming each predicate in module M that appears in the set of predicates denoted by predicates(S).

The operations we wish to define for a modularization system for Prolog are closed inclusion, open inclusion, and union. **Closed inclusion** is defined by the operator $\circ^*$ such that:

$$M[\,[M_1] \circ^* [M_2]\,] =_{def} M[\,[M'] \circ [\,[M_2]\,]$$
$$\text{where } M' = \text{rename}(\,\text{predicates}(M_1 \cap M_2)\,) \circ M_1$$

Closed inclusion is the case where M_1 can see all the definitions in M_2 but M_2 can see nothing in M_1. Also, there must be no symbol conflicts between M_1 and M_2, so we require all the conflicting symbols to be renamed. By closed we mean that module M_1 can not add facts or extend the definitions of clauses in M_2 (M_2 is a closed module).

The second operation we wish to define for gluing modules together is union and is defined by the operator $\circ\cup$ such that:

$$M[\,[M_1] \circ\cup [M_2]\,] =_{def} M[\,[M_1 \cup M_2]\,]$$

Union is the case where all the clauses in M_1 are united with the clauses in M_2, creating a larger database of clauses. M_1 can extend the definitions in M_2 but without affecting M_2's database.

The last operation we wish to define for gluing modules together is open inclusion, defined by the operator $\circ+$ such that:

$$M[\,[M_1] \circ+ [M_2]\,] =_{def} M[\,[M'] \circ\cup [M_2]\,]$$
$$\text{where } M' = \text{rename}(\,\text{predicates}(M_1 \cap M_2)\,) \circ M_1$$

This is the situation where M_1 can see all the definitions in M_2, and M_2 can see all definitions in M_1. The difference between this and closed inclusion is that clauses in M_2 can be extended by adding facts to M_1's database; without altering M_2's database.

Open inclusion is used instead of union when the predicates declared in M_1 must remain distinct from predicates declared in M_2. This produces the restriction that the set of declarations in M_1 intersect with the set of declarations in M_2 is the empty set.

Closed inclusion parallels an abstract data type module. For example, assume we have a module Set containing the type Set, and various predicates that operate on sets (addToSet/3, removeFromeSet/3, createSet/1, etc.). A user of the Set module should not be able to add definitions to a set predicate, nor should the user know the internal representation of a Set (is it a list, or a functor such as set(List)?). Closed inclusion enforces this information hiding.

In contrast, open inclusion would permit the user to extend the definitions of a set. For example, the user may wish to add an additional addToSet/3 predicate that checks for a specific condition. In this way, Set's predicates are extendable. Although, when including the Set module, the system makes sure the including module does not already have an addToSet/3 predicate declaration. This is simply a security measure - the Set module needs to be extended, but notify the user of any conflicts.

3.2.2 Non-logical Predicates

Non-logical predicates interact with a Prolog modularization system in a different manner than conventional predicates. The following is a description of the clause database manipulation predicates assert and retract, the meta predicate prove, the all-solutions predicates, such as bagOf and setOf, and their interaction with Godel's modularization mechanism.

3.2.2.1. Assert and Retract

In procedural languages, module independence is ensured in two ways. First, modules must communicate through the protocol determined by their imports and exports, so the internal objects of a module may not be accessed. Secondly, the contents of each module (code and declarations) is determined statically. That is, no code or declarations are added at run time.

For logic programming languages, the second condition does not hold. At run-time, new clauses can be asserted and old ones can be retracted. This means that a further constraint must be added to a modularization mechanism to ensure module independence. That is, calls to the predicate assert must result in the new clause being added to the local clause-base of the module in which the call appears. In addition, calls to retract must be constrained to remove clauses only from the local clause-base.

3.2.2.2. The Meta-predicate Prove

Meta predicates take on a variety of forms. Some Prologs allow variables to appear as literals in the antecedent of a clause while others do not allow the promotion of logical variables to literals. In the latter case, the system uses a *call*/1 or *prove*/1 predicate. In the former, the logical variable is interpreted as if it where the single parameter to the call or prove predicate. The semantics of these predicates are that the predicate's single parameter is interpreted as though it was inserted directly as a goal in the current refutation.

With a modularized Prolog system, the single parameter *prove* predicate is insufficient. Any call to a predicate must also specify which module the predicate should be proven in. In Godel, the predicate prove(ACall) defaults to proving ACall in the module where

the call predicate originated. To allow a more general scheme, we introduce the predicate prove(AModule, ACall) which is true if AModule can prove ACall. If the predicate prove/2 is used, the name bound to AModule must be visible within the calling module. By visible, we mean it must be a module imported using one of the module gluing operations of union or inclusion. In this way, the prove predicate is restricted since it cannot call predicates from arbitrary modules. If a call to a non-visible predicate is attempted, a run-time error occurs.

3.2.2.3. All Solution Predicates

Second order predicates such as bagOf and setOf are used to find multiple solutions to a query. Their usual parameter sequence is a list of variables, a list of goals, and a list to collect each instance of the first parameter when the list of goals is satisfied. For example, setOf([X, Y], pred(X, Y), AList) is true when AList is a list containing each instance of X and Y such that pred(X, Y) is true.

Conventional implementations of all solutions predicates use the prove predicate. In a modularized Prolog system with well defined communication paths between modules, there are situations where setOf may fail. For instance, suppose setOf is defined in module M_1 and a module M_2 wishes to use it. Suppose also that M_2 calls setOf with its second parameter bound to a predicate P in M_2. Further suppose that M_2 used the closed inclusion operation to "import" setOf. M_2 can see the definitions of setOf and when setOf executes it tries to prove P. However, P is not visible in M_1. It is for these reasons that the use of highly protected abstract data type modules for Prolog are insufficient. The union operation defined earlier remedies this problem by allowing the definition of setOf to be made available to M_2 and all definitions in M_2 to be made available to setOf.

3.2.2.4. Negation as Failure

Godel uses the conventional definition for the **not** predicate. Since not uses the prove predicate, the same restrictions apply to it as to the all solutions predicates. To prove the goal not(p), the predicate p must be visible from the module containing the not predicate. Therefore, any module wishing to use the not predicate must union or openly include not's module.

```
not(X) ⇐
        prove(X) ∧
        cut ∧
        fail.
not(X).
```

3.3. Declarations

Each module in Godel has associated type and clause declarations. A general introduction to typing in logic programming was described in a previous chapter. It encompassed both the declaration of type objects, the types of objects, and the types of parameters to clauses. Here we will give a detailed description of our typing system, outlining the syntactic definitions for typing, and some static semantic errors that our typing system detects.

Several proposals exist to add typing to logic programming (and Prolog). As described earlier, Mycroft and O'Keefe proposed a parameterized type scheme in which type definitions were predicates evaluated at compile time whose failure causes a compilation error instead of a run-time error. [Mishra 1984] considered types as sets of terms generated by a regular tree grammar which describes the objects that a Prolog predicate manipulates. In this way, no explicit type declarations are required.

The availability of type definitions as a means of understanding program structure and behavior is why our approach relies on explicit type declarations instead of type inferencing. We believe that the type definitions in typed Prolog programs can be used in compile time analysis to improve compiled code and execution speeds. Also, typing is useful in debugging - instead of failing with the wrong type argument, it can provide more descriptive run-time and compile time errors. An obvious disadvantage to this approach is the need to specify types. As alluded to earlier, this is favorable during production development, but can be an unwanted burden during prototyping.

We defined four components of a logic language that we wished to type: functions, predicates, constants, and variables. Although the type of a variable can be inferred, there is only one way to type functions, predicates, and constants and that is through declaration statements. A fourth kind of declaration is that of types themselves. These type symbols can then be used by the other declaration statements. By pre-defining type symbols, the problem of referencing an undeclared type during interactive editing is alleviated. The typing of variables is described in section 3.4., **Typing Extensions**. These four language components are typed using the following type statements:

```
<declaration> ::=   <functionDeclaration>     |
                    <predicateDeclaration>    |
                    <constantDeclaration>     |
                    <typeDeclaration>
```

The following are various housekeeping productions used in declaration definitions:

```
<typeList> ::= <type> [ ',' <typeList> ]
<functionTypeList> ::= <functionParameterType> [ ',' <functionTypeList> ]
<constantList> ::= <constant> [ ',' <constantList> ]
<functionParameterType> ::= <constant> | <type>
(<functor>, <predicate>, <constant>) ::= <identifier>[1]
<type> ::= <identifier with first character uppercase>
```

[1] We assume the general definition for <identifier>'s BNF, along with other basic constructs.

3.3.1 Predicate and Function Declarations

There are two types of predicate declarations in Godel. The first type is a simple declaration defining the name of the predicate and the type of its arguments. From these types, the predicate's arity is determined. The second form of predicate declarations appear when the predicate is used as a non-standard compound term. A compound term is a term of the form *<functor> '(' <termList> ')'*, where <termList> is a sequence of terms separated by commas.

Most Prolog systems introduce non-standard compound terms through operators and their associated type and precedence. Operators are only a notational convenience and are used to express terms where the functor is an infix, prefix, or postfix operator. An infix functor appears between its two arguments, a prefix operator appears before its single argument, and a postfix operator appears after its single argument.

To disambiguate the parsing of arguments between the three kinds of functors, precedence numbers are assigned to each argument. In conventional Prolog systems, the type of each functor could be defined as xfx, xfy, and yfx for infix functors, fx and fy for prefix functors, and xf and fy for postfix functors. Here x denotes an argument of lower precedence and y an argument of higher precedence. For example, both operators '+' and '-' have type 'yfx', therefore a-b+c is properly parsed as ((a-b)+c) [which is equivalent to +(-(a,b),c) using compound term notation].

In Godel, we wish to take advantage of declarations, syntax verbosity, and the graphical nature of the environment to combine the declaration of functor types and precedence operators into a single construct. The syntax for an operator predicate is:

```
<predicateDeclaration> ::=   <operatorPredicateDeclaration>   |
                             <standardPredicateDeclaration>   |
                             <externalPredicateDeclaration>.

<operatorPredicateDeclaration> ::=
        PREDICATE <operatorDeclaration> PRECEDENCE <integer>.

<standardPredicateDeclaration> ::= PREDICATE <predicate> UNIFIES WITH <typeList>.

<operatorDeclaration> ::= [<type> <typeSymbol>] <functor> [<typeSymbol> <type>]
<typeSymbol> ::= <= | <
```

Function declarations follow the same format as predicate declarations. There are two types: conventional functor declarations that define the functor's argument types plus the functor's return type, and function declarations that define operators. For operator functors, the precedence field is also included.

```
<functionDeclaration> ::= <operatorFunctionDeclaration> |
                          <standardFunctionDeclaration>.

<operatorFunctionDeclaration> ::=
        FUNCTION <operatorDeclaration> PRECEDENCE <integer> IS <type>.

<standardFunctionDeclaration> ::= FUNCTION <functor> ( <typeList> ) IS <type>.
```

For example, a range function '..'/2 which defines a range of numbers could be defined as [**FUNCTION** Number <=..< Number **PRECEDENCE** 700 **IS** Number] which is equivalent to the evaluable predicate op(700,yfx,'..') with the function declaration [**FUNCTION** '..'(Number, Number) **IS** Number].

3.3.2 External Declarations

External declarations are used to define generic predicates and modules. A typical example is the **quickSort /2** procedure. QuickSort can be defined recursively using a **partition/4** predicate. In turn, the partition predicate uses a comparison predicate to determine the sorting order. Godel's approach to incremental compilation and static semantic checking requires all predicates to be pre-declared. To make a generic

procedure like quickSort, we need to introduce the **external** declaration. Its purpose is merely a placeholder until a module gluing operation merges the formal definition to the actual definition.

<externalPredicateDeclaration> ::= EXTERNAL <predicateDeclaration>

The quickSort example would be coded as:

```
PREDICATE quickSort/2 UNIFIES WITH List, List
PREDICATE partition/4 UNIFIES WITH Object, List, List, List
EXTERNAL PREDICATE compare/2 UNIFIES WITH Object, Object

/* QuickSort code omitted for brevity */

partition(Pivot, [], [], []).
partition(Pivot, [FirstObject|RestToPartition], [FirstObject|PartitionList1], PartitionList2) ⇐
        compare(FirstObject, Pivot) ∧
        partition(Pivot, RestToPartition, PartitionList1, PartitionList2).
partition(Pivot, [FirstObject|RestToPartition], PartitionList1, [FirstObject|PartitionList2) ⇐
        not(compare(FirstObject, Pivot)) ∧
        partition(Pivot, RestToPartition, PartitionList1, PartitionList2).
```

In a module using the partition (or quickSort) predicate, the compare predicate could be defined in that implementing module (for example, compare(X, Y) ⇐ lessThan(X, Y), where lessThan/2 is a "built-in"). The other approach is that compare can be directly aliased[1] to the lessThan predicate.

3.3.3 Type Declarations

Typing systems for Prolog programs have existed for some time. The first implementations were similar to the *lint* program for the C programming language where a "type checker" is executed with a source program as input. The next generation of typing systems added declarations to the Prolog source, so when the

[1] Aliasing is a mechanism described in [Lanovaz, et al.] that allows renaming predicates to avoid name conflicts.

interpreter consulted a source file, these definitions could be read and used to produce "compile time" error and warning messages. We wish to take this a step further by allowing type declarations to exist in the environment during **initial** program entry. This allows automatic detection, feedback, and correction of errors. The following section gives a brief description of existing type inference systems, and introduces the details of Godel's typing system.

Some Prolog type systems use clauses to specify type declarations. For example, to define a list of a certain type, the following clauses could be entered and the system notified that they are for typing:

```
/* list of a specific type */

isListOfType(nil, Type).
isListOfType(cons(Head, Tail), Type) :- Type(Head), isListOfType(Tail, Type).
```

To type the predicate intMember (a member predicate only for integers) using this scheme, the following definitions are needed (where *:- predicateTypeDeclaration* is a compiler directive):

```
:- predicateTypeDeclaration intMember(X, List) :- integer(X), isListOfType(List, integer).
intMember(Item, cons(Item, RestOfList)).
intMember(Item, cons(Top, RestOfList)) :- intMember(Item, RestOfList).
```

Similarly, a notation deviating from Horn clauses could also be employed (similar to O'Keefe's notation):

```
/* list of a specific type */
:- type list(Type) ⇒
        nil |
        cons(Type, list(Type))
```

This specifies that the constant nil is a list of Type, or the function *cons*, which has parameters Type and list of Type, is also a list of Type. The intMember's type declaration would be declared as:

```
:- predicateTypeDeclaration intMember(integer, list(integer))
```

Here *Type* is bound to a built-in type (integer) and is used to define a list of a specific type. This alleviates the need for second order predicates used in the first example by moving the type checking from the clause to the type inference engine.

Godel's approach to typing is slightly different. Aside from the type declarations for predicates and functions, we need the following declarations for constants and types:

```
<constantDeclaration> ::= CONSTANT <constantList> IS <type>
<typeDeclaration> ::= TYPE <identifierList> [ IS <identifier> ]
```

These definitions allow abstract types like Set, List, or Integer to be declared. They also allow constants to be declared. With these declarations and the ones described earlier, we have all that is necessary to produce well-typed Prolog programs.

For example, to well type the intMember predicate, the following declarations would be entered into Godel:

```
TYPE IntegerList
FUNCTION cons(INTEGER,IntegerList) IS IntegerList
CONSTANT nil IS IntegerList

PREDICATE intMember/2 UNIFIES WITH INTEGER,IntegerList
```

The verbosity of these declarations does not impose significant programmer overhead. The use of dialog windows for declaration entry allows the user to type only what is necessary. For example, the function declaration for *cons* is entered into the system by selecting the function declaration menu item in the browser's declaration pane. From there, in-line or standard dialogs are used to enter **only** the function name, its argument types, and its return type (see figures 3.1 and 3.2). When the declaration is accepted

into the system it is displayed in its expanded form. If future changes are needed, only the non-terminal constructs in the declaration are edited.

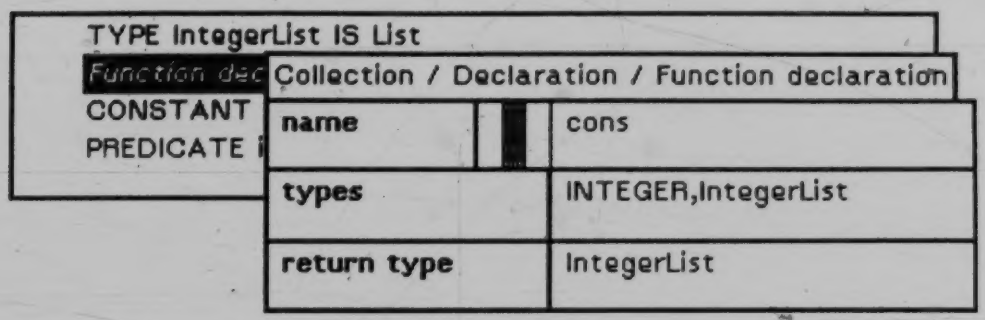


Figure 3.1: Dialog editing of a function declaration.

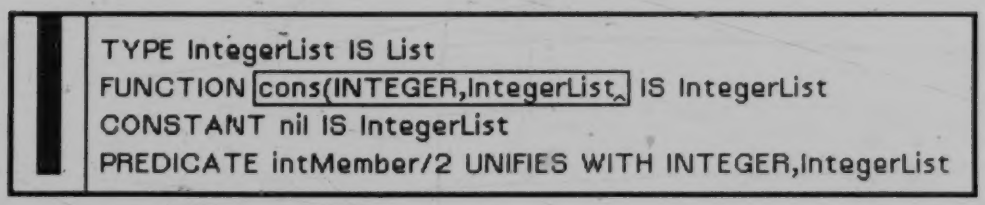

TYPE IntegerList IS List
FUNCTION cons(INTEGER,IntegerList IS IntegerList
CONSTANT nil IS IntegerList
PREDICATE intMember/2 UNIFIES WITH INTEGER,IntegerList

Figure 3.2: In-line dialog editing of a function declaration. The thin-lined border specifies the extent of the editing change. The insertion carat indicates where typed text is to be inserted.

3.3.4 Polymorphic Types

A drawback to the current type inference system is the lack of polymorphic type declarations. For instance, it would be beneficial to declare the function *cons* as a **list of** α, and the predicate intMember to instantiate α to an integer to well type the list of integer.

We take the approach that, for readability and documentation purposes, explicit declarations of **List of** α and **List of** β must be declared as type **ListOf**α and **ListOf**β. However, future versions of Godel could be modified to allow variables to

occur within type declarations. For instance, consider a binary tree data structure defined as:

```
TYPE Tree(TreeType);
FUNCTION tree( Tree(TreeType), TreeType, Tree(TreeType) ) IS Tree(TreeType);
CONSTANT nil IS Tree;
PREDICATE addToTree/3 UNIFIES WITH INTEGER, Tree(INTEGER),Tree(INTEGER);
```

Predicate addToTree(Item, Tree, NewTree) is true if NewTree is Tree with Item added to it. The single parameter to type Tree is instantiated to the built-in type INTEGER in the declaration of addToTree/3 and defines a tree of integers. The difference between this approach and other polymorphic type systems for Prolog is the format of our declarations. We wish to use verbose syntax and contrasting fonts to create a visually appealing and readable format.

3.3.5 Overloading

Overloading is where two functors or constant symbols can have different types. The previous examples overload the constant *nil* which can be either a List or a Tree. In Godel, overloading of functor and constant symbols is not permitted. Godel will issue a warning message since it is our belief that multiply defined symbols usually indicate the program is not properly modularized and an abstract data type facility should be considered. However, Godel could be extended to allow multiply defined symbols in the same module. The type inference system would then check each occurrence of a symbol when determining correct typing.

3.3.6 Additional Type Declarations

Prolog's operational use of first order terms makes them behave as record structures (or type constructors). This idea was proposed in [Ait-Kaci and Nasr 1986] whereby a

term, such as person(X, Y, Z), could be interpreted as a three field record where the programmer assigns meanings to each field (for instance name, age and date of birth). The notion of a functor as a record construct can be used to add further declarations to Godel.

For example, suppose we define an abstract data type for a *person*. Each person knows his name, age, and date of birth. The name is a string, the age an integer, and the date of birth an integer triple of month, day and year. In Prolog, this record could be defined by a functor *person* of arity three, with the formal declaration being the following:

```
TYPE Date;
FUNCTION date(INTEGER, INTEGER, INTEGER) IS Date;

TYPE Person;
FUNCTION person(STRING, INTEGER, Date) IS Person;
```

However, to describe the interpretation for the functors *date* and *person*, we introduce a **record** declaration. This declaration does not change the semantics of Prolog, but only clarifies the programmers intended interpretation. The *date* and *person* records (functions) could be alternately declared as:

```
FUNCTION person IS RECORD
        name            : STRING
        age             : INTEGER
        dateOfBirth     : Date
    ENDRECORD OF Person;
```

The extra identifier declaration fields for records are not needed, but can be used to not only add documentation to programs, but produce more readable error messages. For example, if the functor person was used incorrectly in a clause, a message like "The name of a Person must be of type STRING", or if a field was omitted the message

could read "A Person consists of a name, age and dateOfBirth. Only two fields are present".

Another declaration, a *sub range*, could be added to the declaration list and used to catch compile-time errors and improve program documentation. As with the record declaration, we do not wish a subrange to change the semantics of Prolog. An example of subrange use would be in the Date declaration defined earlier:

```
TYPE Month IS SUBRANGE [1..12];
TYPE Day IS SUBRANGE [1..31];

FUNCTION date IS RECORD
        day     : Day
        month   : Month
        year    : INTEGER
    END RECORD OF Date
```

Our original proposal was that the subrange declaration should not change the semantics of Prolog. It would be useful, however, if the subrange type could be extended to the inference engine so that run-time errors occur if a variable of a subrange type was instantiated to a object (number) outside the defined range. We have not implemented this, but future extensions to Godel may include such operations.

In this section we have described two declarations, subranges and records, that we believe add to the readability and understandability of Prolog programs. They do not change the semantics of Prolog, only strengthen the intended interpretation of a programmer's data structures.

3.4. Typing Extensions

One reason for implementing Godel in Smalltalk-80 was to provide a development regime that permitted experimenting with environment and interpreter changes. As an example, we added type inheritance into Godel's unification mechanism. Adding type

inheritance allowed us to experiment with modifications to both Godel's interpreter and the Prolog language.

The addition of type inheritance was chosen for one major reason. Since we have type definitions of the form **TYPE** A **IS** B and **TYPE** B **IS** C, it seemed natural that we modify the unification algorithm so that objects of type A can unify with objects of type C. The change to the interpreter is simple. Each variable instance carries around another field - its type. When a variable unifies with an object, the two types are coerced to the least upper bound of those types. The variable's type is then set to this upper bound.

Adding type inheritance to Godel required subtle changes to the parser. To declare the type of a variable we extended the syntax of a variable in the terms of a horn clause so that variables can be followed by an identifier representing their type. For example, if we have a clause father(X, Y) such that the variable X is of type Man and the variable Y is of type Person, we can express this fact by father(X : Man, Y : Person).

One example of type inheritance that displayed considerable speed-up in execution time was a Prolog solution to Schubert's Steamroller problem [Szafron and Pelletier 1988] (see Appendix 1 for source listings). A factor of eight in execution time (for this particular problem) resulted from the addition of inheritance into the unification algorithm. We will not go into detail on the benefits of type inheritance in Prolog, as this has been discussed elsewhere.

Similarly, we do not wish to perform a detailed study of the merits of prototyping in Smalltalk-80 versus prototyping in other systems, for this is beyond the scope of this thesis. What we do argue is that the uniform, extendible, and integrated Smalltalk-80

environment simplifies the software modification process. This brief discussion will, we hope, spark the imagination of the reader.

3.5. Static Semantic Errors

This section gives a brief overview of some syntactic and static semantic errors and their corresponding error messages. Several errors are detected by Godel during the programming process. These errors can be grouped into three categories: errors when creating type declarations, type errors when creating clauses, and semantic errors when gluing modules together. The format of these error messages is that the message text appears in normal font. Any context sensitive components are enclosed in angle brackets ('<>') and appear in *italics*.

The following errors can be detected when creating declarations:

[1] The type <*type*> has not been declared in this context.

[2] The *[*<*function*>/<*predicate*>/<*constant*>/<*type*>*]* has already been declared as a <*type*> in this context.

Typing errors encountered when creating clauses are:

[1] The identifier <*identifier*> has not been declared as a type in this context

[2] The variable <*identifier*> has been previously declared as a type <*type*> and is being redeclared as type <*type*>.

[3] Term <*n=1,2,3...*> of predicate *[*<*predicate*> / <*functor*>*]* is declared as a type <*type*> and contains a term of type <*type*>.

[4] The function <*functor*> has arity <*n*> but the formal declaration has arity <*m*>, as declared in the context <*context*>.

[5] The predicate <*predicate*>/<*predicateArity*> has not been declared in this context.

[6] A variable <*variable*> has only been used once in this clause.

Semantic errors that occur when gluing modules are:

[3] Module <*module*> is already included in this context.

[2] The module <*module*> does not exist.

3.6. Summary

This chapter was a detailed description of Godel's language design. We described Godel's modularization system and showed how the gluing operations were designed for building Prolog programs. We also introduced the syntax and semantics of our declarations. Godel's graphical nature allowed us to experiment with various ways of generating self documenting, readable, and informative declaration syntaxes.

We also introduced how new type constructs can be added to both Godel and the Prolog language to clarify the intended interpretation of data structures. We also presented a simple enhancement to the Prolog language and interpreter as an experiment in modifying Godel. Lastly, we demonstrated Godel's "proof reading" capabilities by outlining some static semantic errors.

Object Oriented Environment Design

As Lewis Carol might have put it:

White Knight: *"People think that the Inheritance mechanism of Smalltalk and other Object-Oriented programming languages provides significant support for code level Software Reuse or or ..."*
Alice: *"Or?"*
White Knight: *"Or they don't"*

Neil Haddley - Lancaster, UK

This chapter describes Godel's object-oriented architecture. We first introduce the object-oriented design paradigm and define some terminology used in this chapter. We then introduce the programming environment Guide [Szafron and Wilkerson 1986a] and explain how its architecture was influenced by the object-oriented design methodology. We then describe how Godel could re-use Guide's code by exploiting the class inheritance structure. Lastly, we give a detailed description of the compilation process for Prolog clauses and show how their compiled forms are represented in the Godel environment.

Generally, this chapter is an introduction to the internal details of Godel. It serves as a bridge between the software engineering/user-interface aspects of Godel and the Prolog interpreter (inference engine) described in Chapter six.

4.1. Object-Oriented Programming

Programming languages are classified into "programming paradigms"; a term used to describe a class of programming styles allowing a unique way of representing programmers' intentions. Common programming paradigms include Lisp's function oriented paradigm, Smalltalk's object oriented paradigm, Prolog's logic oriented paradigm, and the common procedure oriented paradigm. When building software systems, the choice of a programming paradigm (and language) will affect how knowledge is expressed. One paradigm may be more natural than another, but this depends on the form of the problem and the developer's view of the problem.

The developer's view when solving a problem is a basic aspect of software design. Four methods commonly used in designing problem solutions are: a control oriented, a data oriented, an object oriented, and a logic oriented method. The control oriented approach to programming analyzes a problem as operations needed to complete a task. The control mechanism, procedural and statement level control structures, are designed first followed by a definition of data-types manipulated by those control structures.

The data oriented approach uses the opposite order. The problem is analyzed to obtain the required data structures, followed by the definition of control structures that will manipulate this data. In contrast, the logic based approach requires statements to be created that describe the data to be manipulated and their relationship. These are taken together to declaratively specify the desired outcome. There is no definition of how to solve the problem (this is left up to the inference engine), only a definition of the problem's logic. Overall, these views of a software system are composed of a collection of data (the information manipulated by the software) and the procedures that manipulate the data (a unit of software).

For example, in a windowing system implemented in a control or data oriented manner, the system would include data representing the size, location, text, and title of each window. The control is represented by procedures to move a window, create a window, and answer if one window overlaps another. The problem with this approach is the data and procedures are treated independently even though they are not.

Object oriented programming rectifies this problem by combining state and behavior into a single entity called an **object**. The notion of an object was first introduced in the late 60's by the language Simula. However, object-oriented programming did not emerge as a new programming paradigm until the creation of Smalltalk in the late 70's. Since then, many language designs for object-oriented programming have emerged. Some of these languages were designed from scratch, such as Trellis/Owl and Eiffel while others were conventional languages with object oriented concepts built on top (hybrid languages) such as C++, Flavors, Loops, and Objective-C. In these languages the object is considered a self contained entity which has: [1] its own private data (state) and [2] a set of operations (behaviors) to manipulate that data.

In the object paradigm an object, like data, can be manipulated, but like a procedure, it also describes how that manipulation is done. To initiate a behavior the system sends a **message** to an object, called the message's **receiver**, to invoke that object's behavior. A message is the name of the behavior (called a selector) possibly with some arguments, and describes what the sender wants to happen, not how it should happen. It is the message's corresponding method, owned by the receiver, which contains executable instructions much like a procedure in a conventional language. A method can only be invoked when an object receives a message whose selector corresponds to that method's selector.

In the windowing example mentioned earlier, an object oriented approach would be to describe windows (opaque, rectangular areas on a display device) as a set of objects - one object for each window on the screen. A window object would have private variables that contain its location or size plus methods describing its behavior of moving itself, displaying itself, erasing itself, or returning its size or location. Each method would have an associated message selector such as **move**, **display**, **delete**, **boundingBox**, and **origin**.

Another feature of object-oriented programming is the concept of a **Class**. A class describes the common features of a collection of related objects. An instance of a class is an object that has its own state. Class descriptions usually include instance variables (the local state for each object) and the methods that manipulate class instances. Therefore, a class is a mechanism to encapsulate the state and behavior of a set of objects.

Classes are related to each other through an inheritance hierarchy. A **subclass** is a specialization of one or more **superclasses**. The subclass inherits state and behavior from its superclasses. The subclass can, however, modify the behavior of its superclasses by redefining a method with the same message selector, *overriding* the superclass's behavior method.

Object-oriented design is considered a powerful representation (or programming methodology) for the construction of a variety of applications, including graphical user interfaces. The object-oriented windowing example is just one formalism for viewing the construction of these interfaces. It is used extensively in the software architecture of many windowing systems and their underlying software components. Prime examples include the MacIntosh's MacApp development environment and the

Smalltalk-80 programming environment. Each of these software systems views the underlying application as a hierarchy of classes, with each hierarchy chain dealing with a specific aspect of the application and interface. By organizing the application as a collection of classes, considerable flexibility between the interaction of the application and the user interface is acquired.

4.2. Guide Architecture

The goal of the Godel project was to create a prototype Prolog programming environment to aid Prolog developers by using a highly interactive and graphical interface. Godel's architecture was strongly influenced by two factors: the Smalltalk-80 language and the Guide project. Object-oriented programming features like classing and inheritance were the driving force behind the abstract data types and hierarchical designs of both Guide and Godel. The ability to create subclasses of existing classes proved extremely useful in reducing both the design and the development time of Godel. To fully understand how this was done, the following sections outline the salient features of Guide's internal architecture and describe how Godel exploited that architecture.

4.2.1. Class Hierarchy

The fundamental building blocks of Guide are:

- The Browsers
    - Environment Browsers
    - Structure Browsers
- The Structures
    - Parse Tree Nodes
    - Symbol Nodes
- The Parser

Browsers are the graphical windows used to view and modify aspects of the Guide environment. The symbol table is a collection of classes used to represent the symbols of a programming language. The parse tree is a collection of classes for representing the internal structure of compiled code. It is used extensively during incremental compilation. The parser is a class that parses the textual representation of parse tree nodes.

The following sections describe the classes that form Guide's building blocks. The notation we use to represent classes and their relationships is borrowed from the Smalltalk-80 language. A class is an identifier starting with an uppercase letter. Following the class name, and in brackets, is a list of instance variables (slots) that belong to that class. Directly under the class name (and indented one tab stop), is the list of all the subclasses of that class. For example, a class **Foo** with instance variables *inst1* and *inst2* is represented as: **Foo** ('inst1' 'inst2').

4.2.1.1. Dependents

Each building block consists of a hierarchy of classes, each hierarchy having the class **GuideObject** as its root. Each object that is a subclass of GuideObject has a set of dependents. The dependents of an object O are any other objects in the system who reference object O. Guide uses dependents to synchronize the existence of entities in the environment. An object in the system can only be removed if it has no dependents, or if each of its dependents is successfully removed. For example, a modeless edit window (dialog) is opened on a structure, S. That structure has the dialog as its dependent. If, for any reason, structure S is removed from the system, its dependent dialog must also be removed. Dependencies are a powerful feature for maintaining consistency within a programming environment.

GuideObject ('dependents')
[. . . other Guide classes . . .]

4.2.2. The Interface

The abstract interface of Godel relies on the Smalltalk class **View** and its various subclasses. Any window on the display screen is an instance of a **View**. Similarly, the language directed editing that Guide supports is modeled by a special subclass of View called a **StructureView.** A StructureView displays the graphical representation of program text. The following section describes the architecture of Guide's windowing system. It first describes the Model-View-Controller (MVC) paradigm used to manipulate windows and then describes the relationship between structures, StructureViews, and language directed editing.

4.2.2.1. MVC Paradigm

The Model-View-Controller paradigm is used extensively in the Smalltalk-80 system when building user-interfaces. As the name suggests, the MVC has three components:

- M - Model: An object that is being represented graphically.
- V - View: An object that presents information from the model to the display.
- C - Controller: An object that reads all user input and updates the model and view accordingly.

For example, a text edit window has as its model an object that, on request from the view, answers the string of text being edited. Its view is an instance of the class **View** which displays the model's text. Its controller is an instance of the class **Controller** that accepts mouse selection events and either positions a cursor at the selection point or, if the mouse is dragged, highlights the selected text. The controller is also responsible for notifying the view and model of other user inputs (for example, keystrokes).

The MVC paradigm is particular only to the Smalltalk-80 PE. However, we are in no way suggesting that this interface paradigm must be used in a production implementation of Godel. The interaction between Godel structures (i.e. Prolog declarations and code) is abstracted enough that conversion to other interface paradigms should be straightforward.

4.2.2.2. Structures

Any object in Guide that can be manipulated using the language directed editor must be an instance of a **GuideStructure**. Each GuideStructure has an instance variable that defines its format. A format is an integer pair that defines the starting and stopping index of the structure's printable representation. The format defines selection and highlight boundaries during language directed editing.

```
GuideStructure ('scope' 'format')
        GuideParseNode ()
        GuideSymbol ()
        [... other structures ...]
```

4.2.2.3. Structure Browsers

To create an interface between the textual representation of programs and language directed editing, Guide defines an abstract class called a **StructureBrowser**. Each StructureBrowser knows the program text it is to manipulate plus the currently chosen structure (*currentStructure*). The selected structure is an instance of a **GuideStructure** or a **StructureListInsertionPoint**. If a GuideStructure is selected, its format defines its highlighting extent. Otherwise, an insertion carat is displayed in the structure edit window (see figure 4.1).

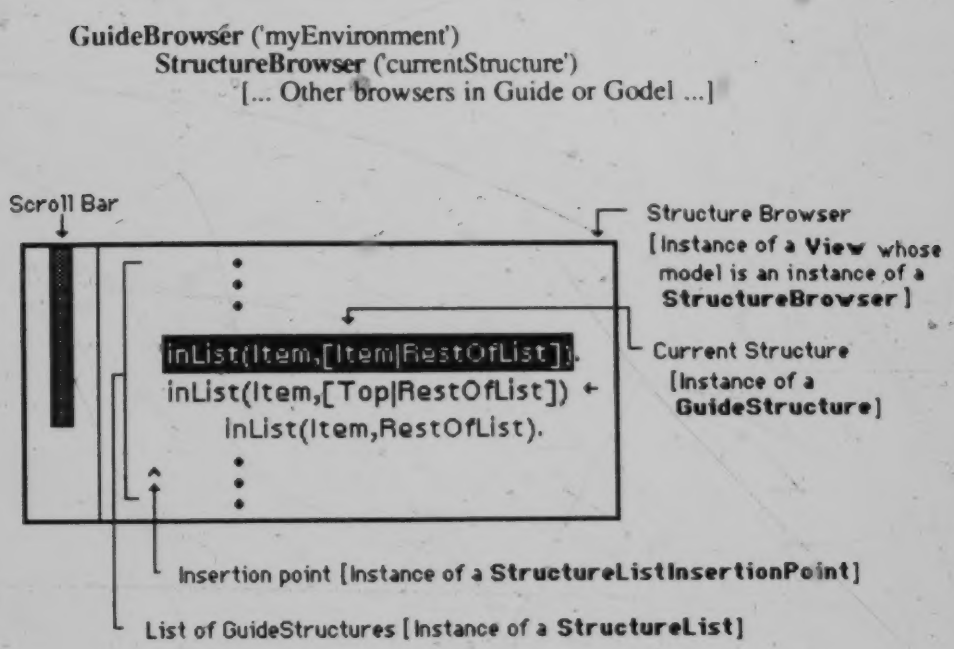


Figure 4.1: Diagram of a Godel structure browser (window). The structures being browsed are compiled Prolog clauses.

4.2.3. Symbol Table

A common structure object in Guide is a symbol. Each symbol represents an identifier in a particular language. For example, in a procedural language a variable name X or procedure name P is an instance of a **GuideSymbol**. GuideSymbols can be classed into declaration symbols (i.e. the variable X is a declared symbol), typed symbols, untyped symbols, or environment symbols.

Declaration, typed, and untyped symbols model standard declarations in procedural languages while environments model various code collections. An environment in Guide is similar to a module or procedure in a procedural language. Environment objects contain information like a table of symbols declared in the environment, a

comment stating the environment's purpose, and a list of sub-environments (similar to nested procedures).

The class hierarchy for symbols is:

```
GuideSymbol ('name' 'isTemplate')
        GuideDeclarationSymbol ('isPublic')
                GuideTypedDeclarationSymbol ('type')
                GuideUntypedDeclarationSymbol ()
        GuideEnvironmentSymbol
                        ('comment' 'symbolTable' 'environmentList')
                GuideContextSymbol ('declarationList')
                GuideProjectSymbol ()
                GuideSystemSymbol ()
```

4.2.4. Parse Tree

The parse tree is the center of Guide's incremental compilation facility. Every program construct is an instance of a **GuideParseNode**. An entire program consists of a collection of parse nodes. In this way, sections of code may be copied, cut, and pasted while maintaining semantic consistency. This enforces that the parse nodes cut or pasted in various environments are semantically sound. Similarly, editing the internal representation of programs maintains syntactic consistency by never permitting ill-formed code fragments to exist.

There are several subclasses of a **GuideParseNode**. The ones of interest are **GuideEditableNodes**, **GuideCompoundNodes**, and **GuideExpressionNodes**. GuideEditableNodes are program nodes that can be interactively edited using dialog boxes. GuideCompoundNodes encapsulate one or more parse nodes. This encapsulation mechanism is how the editor manipulates several nodes as a single structure. Lastly, GuideExpressionNodes form expressions or are part of larger

expressions. For example, an arithmetic expression like (5 * (3 / 6)) is an instance of a GuideExpressionNode.

Overall, parse nodes are used to provide an intermediate internal representation of programs. We have only briefly outlined the key components that form the parse node class hierarchy. We have purposely omitted examples of mappings between program code and parse trees until after discussing Godel's Prolog specific parse nodes.

```
GuideParseNode ()
        GuideEditableNode ('isTemplate')
        GuideExpressionNode ()
                GuideCompoundNode ('body')
```

4.2.5. Parser

The class **GuideParser** parses a stream of characters for a specific, primitive syntactic object. These primitive quantities include standard lexemes like numbers and identifiers. The subclass of a GuideParser is responsible for parsing the language specific tokens and syntax.

An instance of a GuideParser contains two objects: a stream and a controller. The stream is the sequence of characters being parsed. The object returned after a successful parse is an instance of a GuideParseNode which is the intermediate representation of the stream's parsed text. If any errors are detected, the controller is sent a message to display those errors in the program source window.

```
GuideParser ('stream' 'controller')
        [... language specific parser class ... ]
```

4.3. Godel Architecture

We have outlined Guide's class hierarchy . We have also described its main components and how they interact. The following sections describe Godel's

architecture. Particular attention is paid to the notion of subclassing and how Godel exploits Guide's architecture.

4.3.1. Class Hierarchy

In Smalltalk-80, the root of the class hierarchy is the class **Object**. The class Object encapsulates the common behavior of every object in the system. Similarly, the class **GuideObject** defines the common behavior among all Guide classes. We further introduce a class **GodelObject** which encompasses the common behavior of both Godel and Guide Objects. In this way, every object associated with Godel shares a common behavior.

Object
    GodelObject
        GuideObject ('dependents')
            [... other Guide classes ...]
            [... other Godel classes ...]
        [... other Godel classes ...]

4.3.2. Symbol Table

Godel's symbol table contains classes that define the behavior of each symbol in Godel (and the Prolog language). For instance, we have declaration symbols for constants, functions, and types. A constant declaration is modeled by a **GodelConstantSymbol**. This class inherits the behavior of a **GodelTypedDeclarationSymbol** which in turn inherits the behavior of a **GuideDelcarationSymbol**. In this way, a Godel constant object can re-use code previously written for declaration symbols and typed declaration symbols.

Declaration symbols contain various instance variables. Each declaration symbol contains a flag, *isPublic*, which is true if the declaration is visible outside its defining

module. Similarly, each typed declaration symbol has an instance variable *type* which is an instance of a type symbol that specifies the declaration symbol's type.

Type symbols are either user defined, or are one of several pre-defined base type symbols. Among these are a character type, an integer or real type, and an object type. Each of these represent a specific "built-in" type in Godel. For example, assume we have the built-in types CHARACTER and INTEGER which define types for single characters and integers. A predicate asciiValue(ACharacter, AnInteger), which is true if AnInteger is the ascii value of ACharacter, can be declared as [**PREDICATE** asciiValue/2 **UNIFIES WITH** CHARACTER, INTEGER]. Entering the assertion asciiValue('a', 'b') into Godel will produce a compile time error since the character 'b' is not of type INTEGER. The assertion asciiValue('a', 57) is correctly typed.

```
GuideSymbol ('name' 'isTemplate')
    GodelInclusionSymbol ('module' 'symbol')
        GodelUnionSymbol ()
        GodelClosedInclusionSymbol ()
        GodelOpenInclusionSymbol ()
    GuideDeclarationSymbol ()
        GodelPredicateSymbol ('unificationList' 'arity' 'firstCandidate')
        GodelTypedDeclarationSymbol ('type')
            GodelConstantSymbol ()
            GodelFunctionSymbol ('unificationList' 'arity')
            GodelTypeSymbol ()
            GodelVariableSymbol ('index')
        GodelBaseTypeSymbol ()
            GodelCharTypeSymbol ()
            GodelIntegerTypeSymbol ()
            [... rest of the built in types ...]
```

4.3.3. Parse Tree

The parse tree is an internal representation for Godel's compiled source code. Each construct in the source language is compiled into an instance of a GuideParseNode. Each node has a unique behavior which determines the action it must perform at run-time to evaluate itself given a certain run-time context (environment). Before we

describe the run-time behavior of GodelParseNodes, (see chapter six) we must describe the correspondence between each parse node class and its related syntactic construct in a Prolog clause.

Prolog clauses can be grouped into three categories: horn clauses, primitive horn clauses, or assertions. A horn clause is simply a well formed clause with a head and a body. A primitive horn clause is a clause whose body is replaced by source code of a different language. This is usually the language the Prolog interpreter is written - in this case Smalltalk-80. Lastly, an assertion is a clause with no body and represents a fact in the clause database.

We have divided clauses into three categories based on their run-time behavior. Each behavior varies because of the procedural interpretation placed on clauses. For a standard horn clause, if its head can unify with the current goal atom, it must execute each goal in its body. A primitive horn clause executes its body differently since its body is Smalltalk-80 source code instead of a conjunction of atoms. An assertion has no body, so once the unification succeeds, the next goal in the unifying procedure must be attempted (we do not wish to go into details about the interpreter's execution, for this is discussed in Chapter six. We only wish to show that clause parse nodes are created based on their behavior).

```
GuideParseNode ()
    GuideEditableNode ('isTemplate')
        GodelClauseNode ('head' 'numberOfVariables' 'nextClause')
            GodelAssertionNode ()
            GodelHornClauseNode ('tail')
                GodelPrimitiveHornClauseNode ()
                GodelStandardHornClauseNode ('firstCall')
```

In Prolog, the atoms (the heads of clauses and literals in the body) are atomic formula that consist of a predicate and a list of terms. Atoms are a special case of first order

literals in that they cannot be negated. Each atom contains a *nextCall*, an instance variable used for linking together atoms in a clause body. An object called a GodelLiteralAssertionNode represents an atom in the head of a clause, a literal node not needing a nextCall instance.

```
GuideEditableNode ('isTemplate')
        GodelLiteralNode ('predicate' 'termList')
                GodelAtomNode ('nextCall')
                GodelLiteralAssertionNode ()
```

An atom contains a list of terms which in turn are composed of either functions, constants, or variables. The class **GodelFunctionNode** models a function and its arguments while variables are modeled by **GodelVariableSymbols**, encapsulated within a **GuideSymbolNode**. Constants are represented by **GodelAtomicLiteralNodes** or **GodelConstantSymbols**. A **GodelAtomicLiteralNode** encapsulates constants like numbers, characters, or strings while a **GodelConstantSymbol** represents pre-declared constants.

```
GuideExpressionNode ()
        GodelTermNode ()
                GodelFunctionNode ('function' 'termList')
                GodelListNode ('head' 'tail')
        GuideLiteralNode ('literal')
                GodelAtomicLiteralNode ()
        GuideSymbolNode ('symbol')
                GodelSymbolNode ()
```

We have introduced the classes that constitute Godel's parse tree. Here we will show how a Prolog procedure is represented using these classes. Assume we are compiling the procedure **grandfather(X, Y)** which is true if X is the grandfather of Y. This code is represented by the predicate definitions in figure 4.2. These clauses are compiled into an intermediate representation shown in figure 4.3. This figure represents the classes and their instances that form the compiled procedure.

```
TYPE person;
PREDICATE grandfather UNIFIES WITH Person, Person;
PREDICATE father UNIFIES WITH Person, Person;

grandfather(X, Z) ⇐
        father(X, Y) ∧
        father(Y, Z);

/* Included for completeness, but not shown in figure XX */
father(X, Y) ⇐
        parent(X, Y) ∧
        male(X);
```

Figure 4.2: Definition of the grandfather predicate.

4.3.3.1. Parse Tree Optimizations

The previous categorization of syntactic constructs into individual classes is memory inefficient. For example, it is reasonable that a literal assertion node should contain fields (or slots) for each of its terms. The atom **father(X, Z)** would then be represented by a three field object whose first slot is the predicate father, and subsequent slots are the terms X and Y. In our prototype, we wished to distinguish these objects for the following reason. Each object can respond differently to messages sent to it. By having distinct objects, modification of data structures (class definitions) localizes their effects because only the method associated with the modified class needs updating.

In a release version of Godel, the functions and predicate would be succinctly represented as an array for fast argument access. This would diminish the number of object pointer needed to represent Prolog clauses, reducing memory requirements and increasing execution speeds.

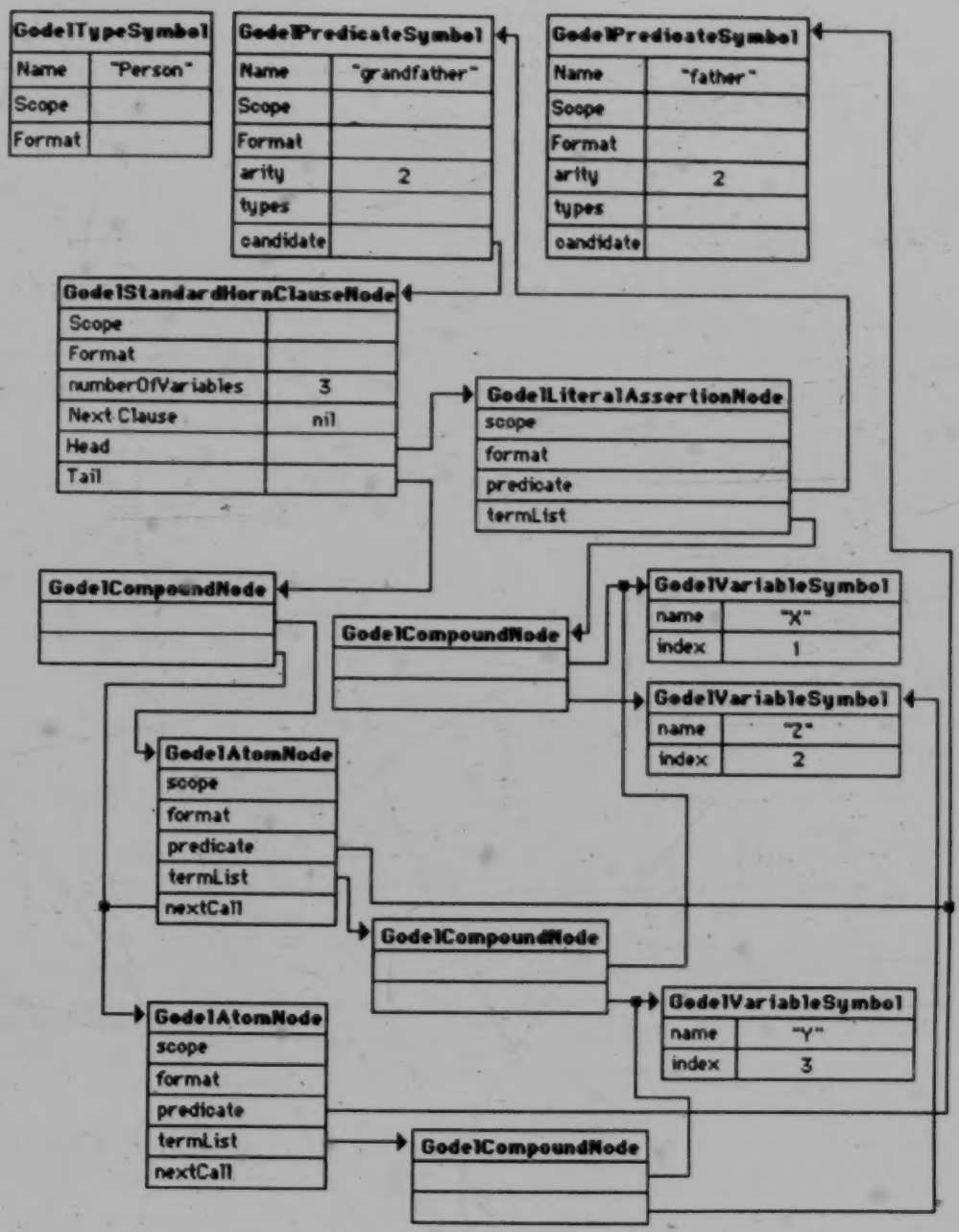


Figure 4.3: Class instances of a compiled Prolog procedure.

4.3.4. Parser

The **GodelParser** is a subclass of a GuideParser. It inherits all lexeme parsing methods defined in the GuideParser. In addition, methods are added to the GodelParser class to parse Prolog specific syntax. Each method corresponds to a BNF production in the language. The parser can parse infix, prefix, and postfix operators as described in section 3.3.1., **Predicate and Function Declarations**. Operators are simply instances of a GodelFunctionNode or GodelAtomNode, depending on their declaration.

During incremental compilation of clauses and declarations, an instance of a Godel parser is created. It parses the text string sent to it by the editor, returning an instance of a GuideStructure. This structure is then sent a message to copy itself in the context it is to be placed into. It is at that time that semantic errors are detected.

4.4. Summary

This chapter was devoted to the description of Guide (a General User Interactive Development Environment) and how Godel could exploit Guide's architecture. We concentrated on Guide's hierarchical design and showed how the object-oriented architecture, classing with inheritance, is extremely useful in designing modifiable and re-usable code. Guide was designed with code reuse in mind. Its classes were abstract enough that subclasses could specialize their superclass's behavior. In this way, Godel could re-use vast amounts of Guide's code base, significantly reducing both the code size and the prototyping time of Godel.

We also described how windows are created using the Smalltalk-80 class View, and briefly introduced the MVC paradigm. Through this, we could introduce Godel's interface architecture. We introduced the concept of a structure and showed how

Prolog clauses are instances of structures. Similarly, we explained how special views called StructureViews enable syntax directed editing of compiled Prolog clauses.

In contrast, the subsequent chapters will describe Prolog's inference engine and show how the object-oriented architecture has produced a different way of designing and coding it.

Prolog Interpretation

To Logic, Hobbes devoted a considerable share of attention.
The peculiarity of his logical system lies in the theory,
that reasoning is merely a numerical calculation.

- J.D. Morell, *Hist. View Philos.* (ed.2) I.i.95, 1874

The basic interpreting algorithm for Prolog was invented by A. Colmerauer and P. Roussel. It was a slow and memory intensive ALOGL-W implementation which was later improved and rewritten in FORTRAN by Battani and Meloni [1973]. Since then, many articles have been written describing various forms of Prolog implementations and more specifically, interpreting algorithms. This chapter is a general description of the main components needed for interpreting Prolog programs and outlines several key optimizations that have emerged during the evolution of these algorithms. This will construct the bridge between the high level descriptions of Prolog interpretation and the intricate details of a specific implementation. With this background, the description of the object-oriented interpreter design will be better understood. This chapter is aimed at a reader unfamiliar with Prolog interpreter details. For those more knowledgeable, they can proceed to chapter six.

The reader should not interpret this chapter as a statement that all Prolog interpreter's are implemented in the manner we are about to describe. There are, however, certain similarities among implementations. It is our goal to summarize these similarities in order to paint a clear picture of the interpretation process. For a more detailed account

of Prolog interpretation and compilation, see [Warren 1977], [Hogger 1984], and [Maier and Warren 1988].

The following clauses represent the knowledge that john is male and the parent of mary, along with the definition of a father as a male parent:

male(john)
parent(john, mary)
father(X, Y) ⇐ parent(X, Y) ∧ male(X).

Given a query, father(Who, mary), the Prolog interpreter's control mechanism (inferencing using linear resolution) infers that john is the father of mary.

A succinct description of this left to right depth first execution strategy appeared in [Cohen 1985]. We will begin by highlighting the more important aspects of his description and use it as a basis for describing the fundamental components of a Prolog interpreter. This algorithm was chosen for describing general interpreter concepts for several reason. First, it is a concise algorithm written in a well known procedural manner. Second, because of its simplicity, it is straightforward to pinpoint areas where optimizations must be made. Third, because of the imperative language used, it is easy to make comparisons between it and interpretive algorithms written in other languages. Lastly, we will compare that design to the object-oriented interpreter described in the next chapter.

5.1. Unification

The variables and terms in the head of a clause are subject to **unification** during the execution of Prolog programs. Unification is one of the fundamental concepts in logic programming. It enables procedures (collections of clauses whose head have the same predicate and arity) to provide input and output parameters for answer extraction. It also

provides clause selection capabilities based on a powerful pattern matching mechanism. The rules for unifying two objects are:

- if both objects are atomic (i.e. numbers, strings, etc), they are compared for equality
- if one object is an unbound variable, it is changed to reference the other object.
- if both objects are unbound variables, the younger one references the older one.
- if either object is a bound variable, it is dereferenced repeatedly.
- if both objects are compound, the functors are compared, and recursively each of the terms are unified (for lists[1], the head and tail are unified).

If any of these tests fail, the unification is said to fail, otherwise a most general unifier (mgu) is returned consisting of a set of variables and the objects they were unified with.

The *occurs check* is an important aspect of the unification algorithm. Consider the case where the goal f(Y, Y) is to unify with f(X, g(X)). The resulting mgu is {X/Y, Y/g(X)}, which simplifies to {X/g(X)}. If we ever tried to print out the binding for X, we get the infinite sequence g(g(g(...))). However, with an occurs check Y would never get bound to g(X).

The occurs check is necessary if unification is to function correctly, but an increase in the complexity of the algorithm from O(n) to $O(n^2)$ results. This is because both objects being unified must be traversed. Therefore, most implementations choose to ignore the occurs check at the sake of an unsound interpreter. There is, however, work being done to eliminate the need for this check. [Plaisted 1984] describes a method for detecting when the occurs check can be avoided by doing global analysis of predicates

[1] Most Prolog implementations have a syntactic sugar for lists. Edinburgh Prolog uses brackets to enclose lists of objects (e.g. [1, 2, 3] is a list of 3 integers, [X|Y] represents a list whose head is the variable X and whose tail is the variable Y) while other Prologs use the "dot" notation (e.g. 1.2.3.NIL). In this paper, the Edinburgh notation will be used.

while work by [Filgueiras 1984] and the creators of Prolog II take a different approach by allowing "infinite structures" to exist.

5.2. Interpreting Algorithm

Unification is a fundamental concept in logic programming, but the workhorse of an interpreter is the resolution inferencing mechanism that acts on the clause database. Prolog's inference engine is significantly less complicated than a general resolution theorem prover due to the strict use of Horn clauses and the execution order. Because of this, a description of the interpreting algorithm is straightforward. Given a query[1] to a prolog system, the Prolog inference engine finds a refutation using the following steps:

[1] **Try a goal**. Search the clause database for a matching clause head.
  [1.1] **If found**: The goal is unified with the head of a clause and recursively each goal in the body is tried (left to right). If all the body's goals succeed, the goal has succeeded. This position in the database is noted.
  [1.2] **Not found**: The search failed, so backtracking to a previous goal with untried candidates is done.
[2] **Re-try a goal**: Reset any variables set by unification and search the clause database from the previous noted position.

Algorithm 1: Pseudo-code interpreter

Assuming a primitive list representation for clauses, the previous example's facts are represented by ((male john)), ((parent john mary)), and ((father X Y) (parent X Y) (male X)). The availability of a function, *head*, that extracts the head of a list and a function, *tail*, that returns a new list consisting of everything but the head of the input list (similar to the *car* and *cdr* functions in Lisp) is also assumed. Also, the function *unify* is an implementation of the previously described unification algorithm. Given the

[1] A query is the same as the body of a clause. It is an existentially quantified list of atoms.

representation of a clause and the above three functions, a simple Pascal-like Prolog interpreting algorithm is presented as algorithm 2.

```
PROCEDURE solve(      listOfGoals    :LIST
                      environment    :LIST;
                      level          :INTEGER )
VAR
    index      :INTEGER
    newenv     :LIST;
BEGIN
IF listOfGoals <> NIL THEN
    FOR index := 1 to n DO
        newenv := unify( copy(head(Rule[index]), level + 1), head(listOfGoals), environment );
        IF newenv <> nil THEN
            solve( append(copy(tail(Rule[index]),level + 1), tail(listOfGoals)),newenv, level + 1 )
        ENDIF
        ENDFOR
ELSE
    printenv(environment)
ENDIF
END solve;
```

Algorithm 2: Prolog interpreting algorithm [Cohen 1985]

The simplicity of this algorithm is due not only to Prolog's simple execution strategy, but also to the algorithms recursive nature. The internal FOR loop corresponds to the first step in the pseudo-code's description - searching the clause database for a matching clause head. A match is found when the function unify returns a non-nil binding environment. This environment is nothing more than a data structure containing variables and their associated bindings, corresponding to the most general unifier described in section 5.1 **Unification**. Thus, the variable *newenv* is set to the new binding environment created during unification. Note that the *unify* function itself needs a copy of the current rule's head, indicated by the function *copy* as *unify*'s first argument. This will distinguish between multiple uses of a recursively defined clause. Similarly, the nesting level parameter (level) of the function *copy* distinguishes multiple uses of the same variable. This is an inefficient example of a technique known as structure copying which will be described in more detail in the next section.

With the current goal successfully unified, a recursive call to *solve* appends the tail of the currently matched clause to the remaining goals (at a new nesting level), and proceeds to interpret the remaining goals (step 1.1. of algorithm 1). If a goal fails, the internal FOR loop exhausts all possible candidate clauses, exits, and returns to the previous recursion level (inside the FOR loop) where the next goal is tried. This automatically takes care of backtracking. In addition, step 2 of algorithm 1 requires resetting of binding information. That is, a backtrack operation occurs when a goal successfully unifies with the head of a clause but the clause's body could not be solved, so the next clause in the procedure must be tried. It is necessary, however, to reset any variables in the goal to the unbound state they were in before unification with the failing clause. Since the above code does explicit copying of literals (using the copy function), variable resetting is not needed.

5.3. Memory Organization

The execution stack (activation records) created by the recursive call to solve in algorithm 2 stores all the necessary information for backtracking and answer extraction. Unfortunately, this naive implementation is too memory inefficient in practice. A means is required to store variable binding and backtrack information in a more efficient manner. Existing Prolog implementations obtain this efficient memory organization using several global stacks.

Of these stacks, the essential one is the **execution** (or frame) stack. Each object on this stack contains enough information to record any bindings that occur during unification. Also, references to other objects on that same stack are recorded and used to determine the correct environment to revert to during a backtrack operation. Since a goal unifying with the head of a clause is analogous to a procedure call, the object that is pushed onto

the execution stack is similar to the activation record created during the recursive call in algorithm 2. If the clause body fails, backtracking occurs, and all the objects created since entering that clause can be popped off the execution stack.

One of the basic components of the objects pushed and popped from the execution stack are variables and their associated bindings. When a goal unifies with a clause head, the most general unifier (binding environment) contains variables and the objects they were bound to. Backtracking requires the resetting of these bindings so that the old state of the computation can be re-created. To remember which variables to reset, the interpreter pushes all variables needing their values reset onto a global **reset** stack, and records in the corresponding frame object on the execution stack the place where these variables reside. During backtracking, a frame object is consulted to find the position in the reset stack that determines the segment of variables whose bindings will be reset.

For example, in figure 5.1 the execution stack is on the left while the reset stack is on the right (each growing upwards). Objects in the execution stack have references to positions in the reset stack. When backtracking to object A, the reset segment is the list of variables from the top of the reset stack to the object A's reset stack reference point.

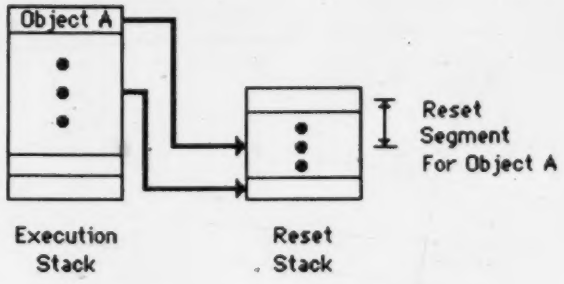


Figure 5.1: Execution and Reset Stacks with Reference Pointer Orientation

One of the goals of early Prolog interpreter design was to minimize the growth of the execution and reset stacks during a refutation. Several optimization techniques were developed to eliminate objects from these stacks (see section 5.4., **Interpreter Optimizations**). To accomplish this, the execution stack was divided into two distinct entities: local and global[1]. The local stack contained binding environment objects whose variable bindings could only reference younger local stack objects (lower in the stack) or objects in the global stack. Likewise, the global stack could not reference objects in the local stack. The reason for this was to guarantee the absence of "dangling" pointers during optimizations. When an object is deleted from a stack due to an optimization, one does not want an object near the bottom of a stack to reference the recently discarded stack object. By orienting reference pointers in a single direction, such problems were eliminated.

This local/global stack structure is only needed when a structure sharing mechanism is used for building terms during unification. The other approach to term construction is known as structure copying. This approach uses a copy stack to store explicit copies of terms, replacing the global stack used in the structure sharing approach. Some believe that structure copying gives a better locality of reference while structure sharing has the potential of severe thrashing in a paged virtual memory system. [Bruynooghe 1982] and [Mellish 1982], however, have compared the relative merits of each of these approaches and no convincing results have emerged.

[1] The two stack representation was developed by David Warren for the DEC-10 PROLOG system at Edinburgh University.

5.4. Interpreter Optimizations

Algorithm 2 is a succinct description of Prolog's inference engine; very inefficient but suitable for explanation purposes. The basic execution strategy can be improved in several ways. The following is a description of several improvements devised to reduce memory requirements and to increase execution time.

Optimizations to reduce memory requirements include last call optimization (LCO), deterministic call optimization (DCO), and deterministic/non-deterministic frame distinctions. Last call optimization is performed when trying to prove the last goal in a clause. If that goal is deterministic (only has one candidate clause), then after unification the newly created node can be copied over the deterministic node on top of the execution stack. The elimination of the extra frame transforms an otherwise recursive call into a memory efficient iterative loop. In some situations this saves considerable amounts of memory.

A deterministic computation is the situation where a Prolog goal has only one possible unifying clause head. Here the interpreter can perform deterministic call optimization; releasing stack frame storage at computation completion. However, this can only be done if there are no backtrack points (alternate clauses) between the start and finish of the computation.

Another common optimization for Prolog interpreters is known as indexing. When the candidate clauses for a goal are being determined by unifying the goal with each clause head in the candidate list, extra information must be recorded if future candidate clauses exist. Naive interpreters like algorithm 2 do not check to see if the subsequent clauses are permissible future candidates. Indexing determines this by matching the first term of the goal with the first term of each remaining clause head in the candidate list. If

future candidates are not possible, there is no need to create a backtrack point. This reduces memory consumption and execution time.

5.5. Summary

In this chapter we have introduced a simple Prolog interpreter. We described both its execution characteristics and its deficiencies. We then introduced some of the optimizations techniques that were developed. We will use this as a basis for the description of our object-oriented interpreter design in the next chapter.

Object-Oriented Interpreter Design

Part of this world of objects, this object-world,
is also part of the very self in question.

G.M. Hopkins, 1880, Sermons & Devotional Writings II.i.127 (1959)

The object oriented programming paradigm is another way of tackling the design and implementation phases of a software project. It has proven useful in designing graphical interactive windowing applications and real-time simulations. To study the effectiveness of the object-oriented design methodology for a graphical interactive Prolog PE, we designed both the interface and the inference engine using the object design method. Our aim was twofold. First, we could use the object paradigm for Godel's graphical interface. Second, we could experiment to see what benefits are gained by implementing a Prolog interpreter in an object-oriented programming language.

The prototype implementation vehicle was the Smalltalk-80 language. We anticipated several advantages by using Smalltalk-80. First, we could utilize Smalltalk-80's garbage collection facility. Garbage collection in Prolog is not a simple task [Bruynooghe 1984]. If we itemize the objects needed by the Prolog interpreter, we can let the Smalltalk garbage collector do our work. Second, the graphical interface of Smalltalk-80 is sufficiently advanced and abstract to allow for experimenting with a variety of interface techniques. We have already demonstrated how Godel's syntax

directed editing takes advantage of this. Experimentation with graphical debugging techniques should also benefit. Third, there has been some work on integrating Prolog into existing programming environments. Some of these include the Interlisp environment [Kahn and Carlsson 1984], and the Nial environment [Jenkins 1987]. Future versions of Godel can be aimed at integrating Prolog into the Smalltalk-80 environment.

In this chapter we describe the object-oriented design of Godel's interpreter. We itemize the objects needed to interpret the Prolog language and describe Godel's memory organization.

6.1. Windowing Environments

In a graphical windowing environment, it is not unusual that several windows view various aspects of source code and execution states. In a non-graphical line-oriented environment, the freedom to browse between contexts (windows) is severely restricted. A natural extension is to move from a line-oriented screen to a raster display containing process windows. Each window houses a particular process like a text editor or an executing interpreter[1]. In terms of an integrated programming environment, there are two problems with this approach. First, the processes cannot communicate by sharing memory (i.e. side effects). Second, code changes are difficult to synchronize.

For example, suppose four windows appear on a display. Two windows contain text editor processes, with each window viewing a different part of a large source code file. The other two windows contain an executing interpreter. A problem arises when changes are made in both text windows, but only one window's changes are

[1] Similar to Sun Corporations *Sunview* environment.

transferred to the interpreter process window. In this case, changes are lost. Similarly, having several interpreter processes running concurrently makes updating the interpreter's program base difficult.

Godel's approach is different than the window-process solution. We have at our disposal a central repository of program clauses. Similarly, we have an editor and inference engine that can manipulate the central repository. There is no reason to restrict the number of queries that can be executed at one time. One window may contain a refutation at midpoint, while a second window contains a completed query. Also, there is no need to reconsult text files when source changes are made because modifications affect the shared repository. For Prolog, the ability to have multiple queries executing concurrently allows a more natural communication pattern between user and environment.

For example, suppose one is debugging a *quickSort* predicate that calls an *append* predicate. Suppose the user is stepping through *quickSort* and learns *append* is not functioning properly. The code browser can be used to edit changes to *append*, and actually test append while quickSort is suspended. When append is operational, the user jumps back to the quickSort trace and continues on.

6.2. Memory Organization: Interpreter Objects

How can several refutations exist at one time? In chapter five we introduced the memory organization of Prolog interpreters. The execution stacks are typically allocated fixed amounts of storage. When allocated storage is exhausted, a stack overflow error is issued. Therefore, to have more than one refutation executing at a time, each refutation must be allocated an address space and a set of stacks. On a

virtual memory system allocating different stacks for each refutation is possible, but communication is inhibited.

To accomplish the goals of a unified environment that permits multiple queries to exist, we adopt the memory management technique of the Smalltalk-80 PE. We view each interpreter data structure as an object that is allocated space via a memory manager.

6.2.1. The Object Pool

In terms of main memory and secondary storage organizations, the use of an object pool as an alternative to paged virtual memory has received some attention [Krasner 1983]. In this approach, objects rather than fixed sized pages are transferred on demand between main storage and secondary storage. There are two advantages to this approach. First, there is a direct relationship between the object-oriented programming paradigm and the object memory. Second, there is the possibility for reducing secondary storage transfer bandwidth because of the absence of unused page segments during object transfer. A criticism of object paging is the overhead associated with maintaining object references. That is, the size of an object cannot be too small or too large.

The problem of determining optimal object sizes is significant in the design of a Prolog interpreter. There are many entities that can be classed as an object such as stack frames, variables, and clauses. The following is an itemization of interpreter objects and Godel's resulting memory organization.

6.2.2. Interpreter Objects

There are three classes that define the objects used in Godel's interpreter: clauses, frames, and binding environments. The classes that define clauses were described in

chapter four and are instances of **GodelParseNodes**. They represent the compiled representation of Prolog clauses. Each clause object responds to execution messages sent to it. Frames are the "activation records" described in chapter five. They represent the current refutation state for a query. Lastly, a **GodelBindingEnvironment** is an object that models the behavior of a collection of variable bindings. It is used when variables are bound to objects during the unification process.

6.2.2.1. Frames

Godel divides frames into three distinct classes: a **GodelFrame**, a **GodelDeterministicFrame**, and a **GodelNonDeterministicFrame**. A GodelDeterministicFrame is added to the run-time stack when a goal successfully unifies with the head of a deterministic clause (no other choice points exist). A GodelNonDeterministicFrame is created when a goal unifies with the head of a clause and other candidates exist. In this case, backtrack information needs to be stored to remember those candidates.

The final frame, a GodelFrame, is used by the procedural debugger. It is nothing but a placeholder for the representation of the run-time stack within the debugger window. For example, we have the predicate **grandfather(X, Z)** $\Leftarrow$ **father(X, Y)** $\wedge$ **father(Y, Z)** which is true if X is the grandfather of Z. We also have the assertions **father(joseph, denis)** and **father(denis, daniel)**. The following frames would be created for the query grandfather(X, Y). When grandfather(X, Y) unifies with grandfather(X, Z), a deterministic frame is added to the stack. When the first goal of the grandfather procedure (father(X, Y)) is called, it unifies with father(joseph, daniel), adding a non-deterministic frame to the run-time stack (because other father predicates exist). When the user steps once more, control returns to the second goal in the grandfather

procedure (father(Y, Z)). Here a GodelFrame is pushed onto the stack. In this way, the system knows which clause to highlight in the code pane of the debugger window, permitting the user to easily view the refutation's history.

GodelFrame ('parentFrame' 'variables' 'scope' 'level')
GodelDeterministicFrame ('return')
GodelNonDeterministicFrame
('nextCandidate' 'previousBacktrack' 'resetVariables')

Each GodelFrame contains four instance variables: *parentFrame*, *variables*, *scope*, and *level*. The *parentFrame* is an instance of a GodelFrame. It is a backpointer that indicates the position in the execution stack that control is passed to when the current goal is satisfied. We call the situation when a goal is satisfied and the execution stack must be popped a *success exit*. The next instance variable, *variables*, is an instance of a **GodelBindingEnvironment**. It contains the variables of the most recently created stack frame. *Scope* is the atom (goal) in the current clause that created this frame, and is used by the graphical debugger for clause highlighting.

The last instance variable is *level*. It is an integer that defines the nesting level of the current frame. In a stack based Prolog interpreter, absolute frame addresses are compared to determine if one frame is lower in the stack than another. To allow for an object based memory architecture that allows multiple executing queries, each frame contains a level (or frame depth). Level comparisons occur when determining which variables to push onto the reset stack (i.e. variables whose bindings need resetting during a backtrack operation). The advantage of this approach is that the interpreter can continue executing until all memory is exhausted. Also, this allows the memory manager to compact memory by moving about frame objects. An obvious disadvantage is the extra word of memory needed for each frame object.

6.2.2.2. Binding Environments

A **GodelBindingEnvironment** is an object that encapsulates a set of molecules. Molecules are binding-context pairs representing run-time variable bindings, and are needed for Godel's structure sharing term representation. A binding environment responds to messages that retrieve or set a variable's binding. Some Prolog interpreters divide variables into local and global. A local variable is allocated space on the run-time stack while a global variable is allocated space on the global stack. In this way, more variable space can be retrieved during a success exit. In Godel, we make no such distinctions. All variables are allocated in the binding environment local to each stack frame. However, future versions could be extended to incorporate this space saving mechanisms.

6.2.3. The Interpreter

Godel's object-oriented architecture allows for the distribution of code among the objects to be executed, in this case, clauses. The code to interpret clauses is associated with each clause object (or parse node). This is in contrast to interpreters using procedural designs where there exists a central procedure that, given a clause base and a query, defines the actions to perform the refutation.

In Godel, there is no central interpreting procedure. Each object in the system knows what to do when receiving a message to execute itself in a given context. The defining context for Prolog interpretation is represented by the class **GodelInterpreter**. A GodelInterpreter is an abstract data type containing methods that set and return the values of its instance variables. It also contains methods to answer queries from the debugging window about the interpreter's current state. Overall, a GodelInterpreter is

the encapsulation of state variables. By having several instances of a GodelInterpreter, several concurrent queries can be executing.

Each GodelInterpreter contains five instance variables: a *currentFrame*, a *currentCall*, a *nextCandidate*, and a *mostRecentBacktrack*. The *currentFrame* points to the top execution stack object. The *currentCall* is an instance of a GodelAtomNode and is the currently executing goal in the refutation.The *nextCandidate* is an instance of a GodelClauseNode and is the next clause in a procedure that *currentCall* will attempt unification. The *mostRecentBacktrack* an instance of a GodelFrame and is set to the most recent non-deterministic run-time stack frame.

CurrentCall is the pivot point when searching for a possible unifying clause heads. The list of clauses that can possibly unify with *currentCall* (the candidate list) are searched sequentially until a clause head successfully unifies with *currentCall*. At this point, *nextCandidate* is set to the clause immediately following that matched clause. If there is a next candidate, *mostRecentBacktrack* is set to point to the topmost execution stack frame. This enables the interpreter to return to this state during backtracking.

6.3. Interpretation

There are two fundamental components of Prolog interpretation: unification and clause selection. The following sections describe the messages sent to various Godel objects to unify terms together, and to select clauses from the clause database for unification during the refutation process.

6.3.1. Unification

The unification process described in chapter five finds the most general unifier (mgu) between two lists of terms. For example, the term f(g(X), X) unifies with f(Y, 1)

producing the mgu {Y/g(X), X/1}. In Godel, each term is represented by a distinct object. Given the representation of variables in a clause as molecules in a binding environment, each term object must respond to the message **#in:** ***Frame*** **unifierFor:** ***anObject*** **in:** ***anObjectsFrame***, where *aFrame* is the message receiver's environment, and *anObjectsFrame* is the environment *anObject* is defined in.

For example, suppose we have the assertion father(denis, daniel). Suppose further that we issue the query father(X, Y). Upon accepting this query, the interpreter creates an environment frame f_1. The interpreter then pushes a temporary frame f_2 onto the execution stack and tries to unify father(X, Y) with the candidate clause father(denis, daniel). To do this, the interpreter sends the message [**in:** f_1 **unifierFor:** [father(denis,daniel)] **in:** f_2] to the atom [father(X, Y)]. If the unification fails the object *False* is returned, otherwise *True* is returned and a global unifier (an instance of class Unifier) is set to the collection of bound variables. In this case, {X/denis, Y/daniel}. Example unification methods appear in figure 6.1. Note that Godel does not perform the occurs check.

```
<GodelClauseNode> in: aFrame unifierFor: anObject in: anObjectsFrame
        "Answer true if my head can unify with anObject, otherwise
         answer false."

        ^head in: aFrame unifierFor: anObject in: anObjectsFrame

<GodelAtomNode> in: aFrame unifierFor: anObject in: anObjectsFrame
        "Answer true if I can unify with anObject, otherwise answer false.
         I can unify with anObject if we have the same predicate name and
        if our terms can unify"

        (anObject isKindOf: GodelAtomNode)
            ifTrue:
                [(predicate unifiesWith: anObject)
                    ifTrue:
                        [termList isNil
                            ifTrue: [^true].
                        ^termList
                                in: aFrame
                                unifierFor: anObject terms
                                in: anObjectsFrame]].
        ^false
```

Figure 6.1: Example unification methods for Godel parse nodes.

There are three advantages to the object-oriented approach to unification. First, the distribution of code among term objects enables the abstraction of each object's behavior (a constant object unifies with another object differently then a variable would). Second, the ability to experiment with new object types without changing the existing interpreter. Suppose a new object is added to the language, for example an array object. When creating the class that describes the array's behavior one just adds a unification method that defines how the array is to unify with other objects.

The third advantage of the object approach to unification is the ability for Godel to use the existing Smalltalk object pool. Smalltalk comes equipped with many pre-defined classes. The classes are structured in a tree rooted at the class **Object**. By adding unification methods to the class Object and its subclasses, each Smalltalk object can respond to unification messages from Prolog structures. Also, since we have defined

primitive clauses (see section 6.3.2.1., **Primitives**) whose bodies are Smalltalk-80 source, the user can write clauses that manipulate Smalltalk objects.

For example, Smalltalk's **Collection** classes encapsulate sequences of objects. It is possible to write a predicate **addToList(AnObject, AList)** where AList is an instance of a Collection instead of a Prolog list structure. Similarly, the class **View** can be used to create windows that Prolog code writes (or draws) in, providing a graphical interface for the Prolog language. Overall, the combination of Smalltalk classes and primitive clauses enables Godel to inherit a sophisticated and powerful environment.

6.3.2. Clause Selection

Prolog clauses are represented by instances of the class **GodelHornClauseNode** or **GodelLiteralNode**. A **GodelClauseNode** has subclasses **GodelPrimitiveHornClauseNode** and **GodelStandardHornClauseNode**, each of which specializes the behavior of the abstract class **GodelClauseNode**. By separating Prolog language constructs into classes that model the construct's behavior, we were able to distribute the inference engine among these classes.

Each clause class can respond to the message **#stepIn:** ***anInterpreter***. The single argument, *anInterpreter*, is an instance of a GodelInterpreter. When a horn clause receives the **#stepIn:** message, it performs all necessary operations to execute itself in the context determined by the single argument, *anInterpreter*. The value it returns indicates whether it failed, succeeded, or succeeded and results are to be displayed.

For example, suppose we have the grandfather and father predicates as defined previously, along with the query grandfather(X, Y). The interpreter's currentCall is the GodelAtomNode, grandfather(X, Y). The message **#stepIn** is sent to the node

grandfather(X, Y). A GodelAtomNode's behavior upon receiving this message is to search for a procedure whose predicate and arity matches its predicate and arity. When a clause is found, that clause's head must be unified with current goal atom. If this succeeds, the interpreter updates the run-time stack and sets the currentCall to be the new clause's first call (the first atom in its body). The **#stepIn:** message is again sent to the new currentCall, father(X, Y). This process continues until the refutation's success or failure.

6.3.2.1 Primitives

Primitive clauses are special clauses whose body are not a conjunct of atoms, but code written in a different language (usually the interpreter's implementation language). In Godel's case, this is Smalltalk-80 code. There are two advantages to readily accessible and modifiable primitive clause code. First, existing primitive methods can be specifically tailored by Godel's users. Second, by using Smalltalk-80's incremental compilation facility, primitive clauses can be freely added to a Prolog application. Although adding primitive clauses to Prolog applications detracts from the elegance of having one implementation language, it can be offset by the gains in efficiency. For an example of primitive clauses, see figure 6.2.

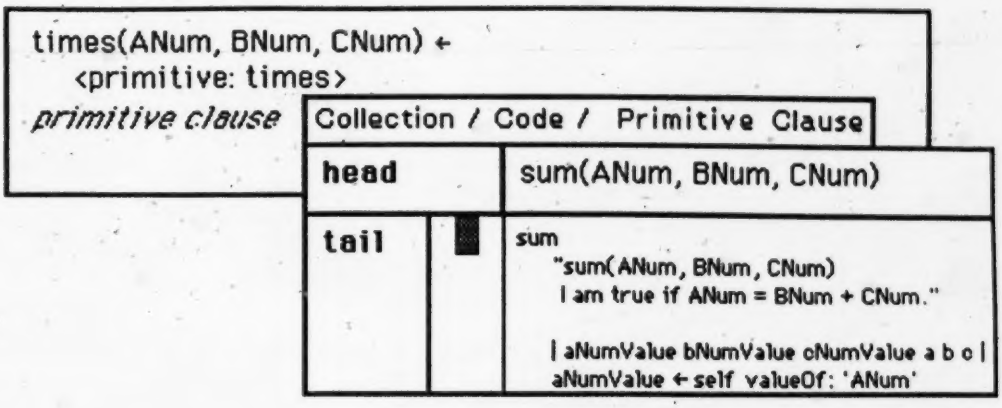
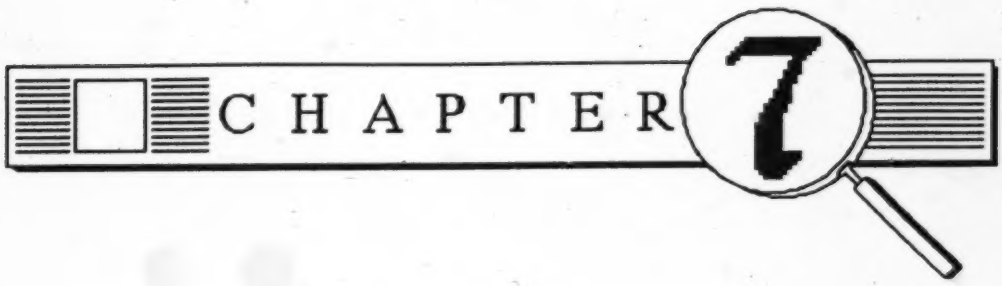


Figure 6.2: Primitive clauses. The *times* primitive is already defined. The sum primitive is currently being edited within a dialog window. Note the Smalltalk code in the body of the clause.

Associated with primitive methods are a collection of methods used to retrieve and set the values of the primitive clause's instance variables. For example, the sum primitive sends the message **#valueOf:** ***aVariableName*** to self (i.e. the object who owns the sum primitive method) to retrieve the value of the single parameter *aVariableName*. Similarly, the message **#valueOf:** ***aVariableName*** **put:** ***anObject*** binds to the variable *aVariableName* the object *anObject*. These messages, along with the specification of Prolog term structures as distinct classes with well defined message interfaces, provides the developer with the necessary tools for manipulating arbitrary Prolog terms.

6.4 Summary

We have designed Godel's interpreter in an object oriented manner. We have itemized the objects required for interpreting the Prolog language. These include the frames, the binding environments, the interpreter abstraction, and the program clauses. We have also outlined Godel's memory organization. In the next chapter, we summarize the results learned from designing, implementing, and using Godel.



Conclusion

Concepts without factual context are empty;
sense data without concepts are blind . . .
The understanding cannot see. The sense cannot think.
By their union only can knowledge be produced.

Immanuel Kant 1724-1804

Godel is an experiment in creating a user-interactive (graphically oriented) Prolog programming environment. Our study spanned a wide spectrum; from modularizing Prolog code to developing an interactive debugging formalism. We not only incorporated software engineering features to make Prolog suitable for large program development, but designed and implemented a prototype environment that took advantage of the graphical interactive nature of modern workstations.

In this thesis, we argued for the addition of three components to a Prolog environment: modularization, typing, and open world programming. We demonstrated that, through the use of a graphical interface, new ways of representing program structure and state are possible. We are not claiming that our approach is sacred, only that future work can take advantage and build upon our original ideas. Moreover, we can experiment with the "look and feel" aspects of the environment to suggest enhancements and hopefully guide future work.

In chapter two we described the general architecture of a Prolog PE. We argued that, for incremental compilation of clauses, a modifiable central clause repository is an asset. In the case of modularization, we argued for a module structure geared towards constructing logic programs. We defined the operations of union, open inclusion, closed inclusion, and demonstrated their usefulness. We also outlined Godel's interface features and described how they affect editing clauses, browsing modules, and debugging. In chapter three we introduced our module semantics. We also introduced our type declarations and described the syntax directed editing and graphical representation of these declarations.

Chapter four introduced Godel's object-oriented design. Chapter six expanded on this by describing the interpreter's design. Various work is underway to study the effects of the object-oriented design methodology in designing large software systems. From our description in chapter four, we hope the reader has gained some appreciation of how classing and inheritance promotes code re-use. This can prove invaluable when designing large, modifiable, and extendible systems. However, drawbacks emerge when the object design is used at a finer level.

An example is that of unification. Our unification algorithm's performance can be improved significantly if it were written in a table driven manner as opposed to sending messages to individual objects. When the number of unifiable objects increase, so do the number of messages, and therefore the number of method look-ups. The procedural interpretation for unification seems more appropriate than the object based method. However, the object based approach reveals its usefulness when extending the "kinds" of unifiable objects. Recall the examples of adding an Array type, or utilizing Smalltalk's class base by permitting arbitrary Smalltalk objects to unify with Prolog variables and terms. In this case, no modifications to the interpreter or

unification algorithm occur. The only change is to add behavior methods to the new object that define the actions that object takes to unify itself with other objects.

Graphically oriented interaction paradigms are deemed by most people to require considerable amounts of effort in their design and development. In a practical sense, where high speed, high performance applications are mandatory, this is true. But in a research environment, where experimenting with new ideas, new formalisms for decreasing the communication barrier between user and computer, this is not the case. Smalltalk-80 is an environment that meets this demand. New ideas can be formulated, designed, implemented, and tested far before production development is undertaken. This means that increasingly ambitious ideas can be realized, their utility tested, and results generated, without waiting for technology to close the gap.

As for the interpreter, why the object-oriented design? There are three reasons for this. First, the utility of the object-oriented programming paradigm for use in developing graphical interfaces is well documented. By implementing the interpreter in the object paradigm, we obtained a uniform architecture between the graphical interface and the inference engine. This permitted the design of a highly integrated interpreter and debugger.

Second, we wanted to experiment with the object design of a Prolog interpreter. We conclude that only cosmetic differences exist between the object-oriented interpreter and a conventional interpreter. The message sending paradigm of Smalltalk permitted us to distribute the inference engine among the various clause objects. However, the basic execution strategy of Prolog must remain the same. The execution stack must be present, variable bindings must be set, and backtrack points must be recorded. Even when it comes to optimizations, the object-oriented approach ultimately performs the

same task as a conventional interpreter. We are not saying the two designs are identical. There are subtle differences (i.e. the distributed code) that permit Godel's interpreter to behavior suitably in a user-interactive environment.

Third, experimentation is needed between logic and user interfaces. In [Grossman and Ege 1987] they formulate a system where the user interface management system uses logic to declaratively specify and control the ways that objects are composed to create interfaces. Their system relies on two components. An interpreter with intelligent backtracking capabilities (for undo operations), and a system to display these graphical objects. We believe Godel can provide a foundation for further work in this area.

Godel is an experimental test-bed for many issues: incremental compilation, graphical debugging, modularization, and interpretation of Prolog programs. We have, however, only scratched the surface of many of these issues. Because of this experimental nature, we argued for its prototype development using the Smalltalk-80 programming environment.

Graphical debugging is one area that can benefit. In chapter two, we introduced a prototype graphical trace and spy debugger. Although this debugger is currently implemented for conventional Prolog execution, it can prove invaluable in debugging a Prolog implementation that uses a more flexible execution strategy. For example, [Naish 1985a and 1985b] describes a Prolog system (NU-Prolog) that incorporates co-routining directly into the inference engine. Co-routining was added to Prolog as a step towards building a pure logic language (not requiring non-logical predicates like cut and var).

This flexible execution strategy makes it very difficult to follow a program's movements, as goals are delayed if their input is not sufficiently instantiated. With a

graphical debugger, contrasting backgrounds and fonts can help to indicate delayed predicates, executing goals, and succeeded goals. However, much work needs to be done to experiment with these ideas. Godel provides such an experimental vehicle.

Although the Smalltalk environment permitted us to experiment with enhancing the usability of a Prolog PE through a graphical interface, there are performance issues to contend with. Here we suggest three ways of improving Godel's performance. The first is compiling Prolog clauses directly into Smalltalk-80 source. This has the advantage that the Smalltalk-80 environment tools (browser and debugger) can be used in debugging the compilation and execution process. A drawback would be the difficulty in interfacing the compiled code with Godel's graphical debugger.

The second approach to increasing performance is to experiment with Smalltalk's user primitives. User primitives are extensions to Smalltalk's virtual machine (the Smalltalk interpreter). An interface from the environment to this machine enables the developer to implement "high performance" methods in assembly language or C. For example, unification is a computationally expensive operation. If the unification methods were taken out of Smalltalk and placed into primitive methods, considerable performance increases would occur. A drawback is the extendability and modifiability of Godel's interpreter is lost.

The last approach is to compile the Smalltalk implementation. Work by [Atkinson 1986] has demonstrated that compiling a typed Smalltalk language is possible, where a significant increase in execution time was observed. A second approach is to transform the Smalltalk implementation into another compilable object-oriented language. [Cox and Schmucker 1987] have designed a tool to convert Smalltalk-80 code into Objective-

C. Such a tool would prove invaluable in converting Godel from a prototype to production state.

"With inadequate tools, even the best craftsperson must struggle to produce good work."

Bibliography

Ait-Kaci, H. and Nasr R. [1986] LOGIN: A Logic Programming Language with Built-in Inheritance, *Journal of Logic Programming* 1986:3:185-215.

Apt, K.R., van Emden H.M. [1982] Contributions to the Theory of Logic Programming, *J. of ACM* 29(3) (1982) 841-862.

Atkinson R.G. [1986] Hurricane: An Optimizing Compiler for Smalltalk, OOPSLA Conference Proceedings, 1986, pp. 151-158.

Battani, G. and Meloni H. [1973] Interpreteur du language de programmation PROLOG. Research Report, Artificial Intelligence Goup. Univ. of Aix-Marseille, Luminy, France.

Bruynooghe, M. [1982] The Memory Management of Prolog Implementations. In *Logic Programming*: Clark, K.L., Tarnlund, S.A. (ed.). Academic Press.

Bruynooghe, M. [1984] Garbage collection in Prolog interpreters, in *Implementations of PROLOG*, Cambell J.A. (ed.), Ellis Horwood Ltd., 1984, pp. 250-258.

Buxton B. [1987] The "Natural" Language of Interaction: A Perspective on Non-Verbal Dialogues, *CIPS Edmonton Intelligence Integration Proceedings*, 1987, pp. 311-316.

Cheng, M.H. [1984] *Design and Implementation of the Waterloo Unix Prolog Environment*. M.Math. Thesis. Univ. of Waterloo, Ontario, Canada.

Chomicki J. and Minsky N.H. [1985] Towards a Programming Environment for Large Prolog Programs, IEEE 1985, p. 230-240.

Clark, K.L. and McCabe F.G. [1981], IC-Prolog Language Features, *Logic Programming*, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 253-266.

Clark, K.L. and McCabe F. [1984] *micro-PROLOG: Programming in Logic*. Prentice-Hall.

Clocksin, W. and Mellish C. [1981] *Programming in Prolog*. Springer-Verlag.

Cohen J. [1985] Describing Prolog By Its Interpretation and Compilation, *Communications of the ACM*, Dec. 1985, Vol. 28, No. 12.

Cox B.J., Schmucker K.J. [1987] Producer: A Tool for Translating Smalltalk-80 to Objective-C, *OOPSLA Conference Proceedings*, 1987, pp. 423-429.

Delisle N.M., Menicosy D.E. and Schwartz M.D. [1984] Viewing a Programming Environment as a Single Tool, *ACM*, 1984.

Filgueiras M. [1984] A Prolog interpreter working with infinite terms, in *Implementations of PROLOG*, Cambell J.A. (ed.), Ellis Horwood Ltd., 1984, pp. 259-267.

Francez N., S. Goldenberg, R.Y. Pinter, M. Tiomkin and Tsur S. [1985] An Environment for Logic Programming, *ACM SIGPLAN 85 Symposium on Lang. Issues in Prog. Env.*, Vol. 20, No. 7, 1985.

Gallaire, H. and Minker J. [1978] *Logic and Data Bases*, Plenum Press, New York.

Goguen J.A. and Meseguer J. [1984] Equality, Types, Modules, and (Why Not?) Generics for Logic Programming, Journal of Logic Programming 1984:2:179-210

Goldberg, A. and Robson D. [1983] *Smalltalk-80: The Language and its Implementation*, Addison Wesley.

Grossman N., Ege R.K. [1987] Logical Composition of Object-Oriented Interfaces, *OOPSLA Conference Proceedings*, 1987.

Hodges, W. [1971] *Logic.* Penguin Books.

Hogger C.J. [1984] *Introduction to Logic Programming*, Academic Press, 1984.

ICOT [1982] *Outline of Research and Development Plans for FGCS*, ICOT Research Center, Minato-ku, Tokyo, May.

Inside Macintosh, Vol. I, II, III [1985], Adison-Wesley, Inc., Reading, second printing, 1985.

Jenkins, M. [1987] Intelligent Data Bases & NIAL, in *Expert - The Magazine for the Artificial Intelligence Community*, Vol. 2.

Kahn K.M and Carlsson M. [1984] How to Implement Prolog on a LISP Machine, in *Implementations of PROLOG*, Cambell J.A. (ed.), Ellis Horwood Ltd., 1984.

Kleene, S. [1967] *Mathematical Logic.* John Wiley & Sons.

Komorowski, H. and Omori S. [1985] A Model and an Implementation of a Logic Programming Environment, *Proceedings ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, (Seattle; June), pp. 191- 198, 1985.

Kowalski, R. [1979] *Logic for Problem Solving.* Elsevier North Holland.

Krasner G. (ed.) [1983] *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley.

Lanovaz D., Szafron D. and Wilkerson B. [1987] The Synergism of Logic-Based Programming and Software Engineering: A Programming Environment Approach, CIPS Edmonton Proceedings, 1987, pp. 43-53.

Laursen J., Atkinson R. [1987] Apus: A Smalltalk Production System, *OOPSLA Conerence Proceedings, 1987*, pp. 377-387.

Lloyd, J.W. [1984] *Foundations of Logic Programming*, Springer-Verlag, 1984.

Maier D. and Warren D.S., [1988] Computing with Logic - Logic Programming With Prolog, Benjamin/Cummings Publishing Company, Inc.

Mellish C. [1982] An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter. In *Logic Programming*: Clark, K.L., Tarnlund, S.A. (ed.). Academic Press, New York, pp. 99-106.

Meyer B., Nerson J.M. and Matsuo M. [1987] Eiffel: Object-Oriented Design for Software Engineering, *Proceedings 1st European Software Engineering Conf.*, Strasbourg, France, Sept. 1987, *Lec. Notes in Comp. Sci. vol. 289*, Springer-Verlag, 1987.

Miller D.A. [1986] A Theory of Modules For Logic Programming, in *IEEE Symposium on Logic Programming*, 1986.

Mills J.W. [1986] A high performance LOW RISC machine for logic programming. *Proceedings of the IEEE 1986 3rd International Symposium on Logic Programming.*

Mishra P. [1984] Towards a theory of types in Prolog, in *Proc. IEEE International Symp. on Logic Programming*, Atlantic City, 1984.

Morishita S. and Masayuki N. [1985] Prolog Computation Model BPM and its debugger PROEDIT2, from *Lecture Notes in Comp. Sci.* No. 264, Eiiti Wada (ed.), pp. 147-158.

Muller H.A., Hoffman D.M., Horspool, R.N. and Levy M.R. [1987], K2: A Software Development Environment for Programming-in-the-large, CIPS Edmonton Proceedings, 1987, pp. 62-27.

Mycroft, A. and O'Keefe R. [1984] A Polymorphic Type System For Prolog, *Artificial Intelligence*, **23**, pp. 295-307, 1984.

Myers, G. [1979] *The Art of Software Testing*, p. 36, Wiley, New York, 1979.

Naish, L. [1985a] Automating Control for Logic Programs. *The Journal of Logic Programming*, VOl.2, No. 3, Oct. 1985.

Naish, L. [1985b] Negation and Control in Prolog. Ph.D. Thesis, University of Melbourne. Also available as *Springer-Verlag Lecture Notes in Computer Science No. 238*, Goos, G. and Hartmais J., (eds.), 1986.

Naish, L. [1986] Negation and quantifiers in NU-Prolog. *Proceedings of the Third International Conference on Logic Programming*. Springer-Verlag 1986.

Numao M. and Jujisaki T. [1983] A Visual Debugger for Prolog, in *Proc. of the IEEE Second Conference on Artificial Intelligence Applications*, pp. 422-427.

O'Keefe, R.A. [1985] Towards an Algebra for Constructing Logic Programs, *Proceedings of the IEEE Symposium on Logic Programming*, 1985, pp. 152-160.

Parnas, D.A. [1972a] A Technique for Software Module Specification with Examples, *Comm. ACM*, **15**(5), May 1972.

Parnas, D.A. [1972b] On the Criteria to be Used in Decomposing Systems into Modules, *Comm. ACM*, **15**(12), December 1972.

Pereira L.M. [1986] Rational Debugging in Logic Programming, *Lecutre Notes in Computer Science*, No. 225, Goos G. and J. Hartmanis (eds.), pp. 203-210.

Plaisted, D. [1984] The Occurs-check Problem in Prolog. *Proc. IEEE 1984 International Symposium on Logic Programming.*

Reiss, S.P. [1984] Graphical Program Development with PECAN Program Development Systems., *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.*

Roberts, G. M. [1977] *An implementation of PROLOG.* M.Sc. Thesis. Univ. of Waterloo, Ontario, Canada.

Robinson, J.A. [1965] *A Machine Oriented Logic Based on the Resolution Principle.*, ACM Journal 12(1), pp. 23-41.

Sergot, H. [1983] A Query-the-User facility for logic programming. *Proc. European Conference on Itegrated Interactive Compuer Systems*: Degano, P. and Sandewall, E., eds. North-Holland.

Shaprio, E. [1982] *Algorithmic Program Debugging*, The MIT Press.

Sterling, L. and Shapiro E. [1986] *The Art of Prolog.* The MIT Press.

Sweet R.E. [1985] "The Mesa Programming Environment", *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments.*

Stroustrup B. [1986] The C++ Programming Language, Addison-Wesley, Reading, c1986.

Szafron D. and Pellterier J. [1988] A Prolog Solution To Schubert's Steamroller Problem, University of Alberta, Canada, To be published.

Szafron D. and Wilkerson B. [1986a] GUIDE: Preliminary User's Manual, Technical Report TR86-05, Department of Computing Science, Univerity of Alberta, Edmonton, March 1986.

Szarfron, D. and Wilkerson B. [1986b] Some Effects of Graphical User Interfaces on Programming Environments, *CIPS Congress 86*, (Vancouver; April), pp. 311-317, 1986

Tick, E. and Warren D.H.D. [1984] Towards a Pipelined Prolog Processor. *Proceedings of the IEEE 1984 International Symposium on Logic Programming*. Also in *New Generation Computing*, 2, Springer-Verlag.

Tietelbaum T. and Reps T. [1981] The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, in *Interactive Programming Environments*, Barstow D.R., Shrobe H.E. and E. Sandwell (eds.), McGraw-Hill, 1984, pp. 97-116.

Teitelman W. and Masinter L. [1981] The Interlisp Programming Environment, in *Interactive Programming Environments*, Barstow D.R., Shrobe H.E. and E. Sandwell (eds.), McGraw-Hill, 1984, pp.83-96.

van Emden H.M. [1982] An Interpreting Algorithm for Prolog programs, in *Implementations of Prolog*, J.A. Campbell (ed.), Ellis Horwood, 1984.

van Emden, M.H. and Kowalski R.A. [1976] The Semantics of Predicate Logic as a Programming Language, JACM, 23, 4 (Oct.), pp. 733-742.

van der Meulen P.S. [1987] INSIST: Interactive Simulation in Smalltalk, *OOPSLA Conf. Proc.*, 1987, N. Meyrowitz (ed.).

Walker A., McCord, M., Sowa, J. and Wilson W. [1987] *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Addison-Wesley.

Warren, D.H.D. [1976] Generating conditional plans and programs, *Proc. AISB Summer Conference*, Edinburgh, pp. 344-354.

Warren, D.H.D. [1977] Implementing PROLOG - compiling predicate logic programs. Research Reprots Nos. 39 and 40, Dept. of Artificial Intelligence. Univ. of Edinburgh, Scotland.

Warren, D.H.D. [1980] An improved PROLOG implementation which optimizes tail recursion, *Proceedings of the Logic Programming Workshop*, July, 1980, Debrecen, Hungary, S.A. Tarnlund (ed.).

Warren D.H.D. [1983] *An Abstract Prolog Instruction Set.*, SRI Technical Note 309, Menlo Park, California.

Winston, P. and Horn B. [1984] *LISP, Second Edition*, Addison-Wesley, 1984.

Appendix 1: Horn Clause Solutions to Schubert's Steamroller

```
wolf(wolf);
fox(fox);
bird(bird);
snail(snail);
grain(grain);
caterpillar(caterpillar);

animal(A) <-
        wolf(A);
animal(A) <-
        fox(A);
animal(A) <-
        bird(A);
animal(A) <-
        caterpillar(A);
animal(A) <-
        snail(A);

plant(A) <-
        grain(A);
plant(food1(A)) <-
        caterpillar(A);
plant(food2(A)) <-
        snail(A);

smaller(A,B) <-
        caterpillar(A) & bird(B);
smaller(A,B) <-
        snail(A) & bird(B);
smaller(A,B) <-
        bird(A) & fox(B);
smaller(A,B) <-
        fox(A) & wolf(B);

doesntEat(A,B) <-
        wolf(A) & fox(B);
doesntEat(A,B) <-
        wolf(A) & grain(B);
doesntEat(A,B) <-
        bird(A) & snail(B);
doesntEat(C,D) <-
        animal(C) & plant(D) & animal(A) & smaller(C,A) & plant(B) &
        doesntEat(A,C) & doesntEat(A,B);

eats(A,B) <-
        bird(A) & caterpillar(B);
eats(A,food1(A)) <-
        caterpillar(A);
eats(A,food2(A)) <-
        snail(A);
```

```
eats(A,C) <-
        animal(A) & animal(C) & smaller(C,A) & plant(D) & eats(C,D) &
        plant(B) & doesntEat(A,B);
eats(A,B) <-
        animal(A) & plant(B) & animal(C) & smaller(C,A) &
        doesntEat(A,C) & plant(D) & eats(C,D);

goal(A,B,C) <-
     animal(A) & animal(B) & grain(C) & eats(A,B) & eats(B,C);
```

Listing 1: A Prolog Solution to Shubert's Steamroller

```
TYPE organism;
TYPE animal IS organism;

TYPE wolf IS animal;
TYPE fox IS animal;
TYPE bird IS animal;
TYPE caterpillar IS animal;
TYPE snail IS animal;
TYPE plant IS organism;
TYPE grain IS plant;
FUNCTION food1(caterpillar) IS plant;
FUNCTION food2(snail) IS plant;

PREDICATE smaller/2 UNIFIES WITH animal, animal;

smaller(X : caterpillar, Y : bird);
smaller(X : snail, Y : bird);
smaller(X : bird, Y : fox);
smaller(X : fox, Y : wolf);

PREDICATE doesntEat/2 UNIFIES WITH animal, organism;

doesntEat(X : wolf, Y : fox).
doesntEat(X : wolf, Y : grain).
doesntEat(X : bird, Y : snail).
doesntEat(C : animal,D : plant) :-
        smaller(C,A : animal),
        doesntEat(A,C),
        doesntEat(A, X : plant).

PREDICATE eats/2 UNIFIES WITH animal, organism;

eats(X : bird, Y : caterpillar).
eats(X : caterpillar, food1(X)).
eats(X : snail, food2(X)).
eats(A : animal, C : animal) :-
        smaller(C, A),
        eats(C, V : plant),
        doesntEat(A,B : plant).
eats(A : animal,B : plant) :-
        smaller(C : animal,A),
        doesntEat(A,C),
        eats(C, V : plant).

PREDICATE goal/3 UNIFIES WITH animal, animal, plant;

goal(A, B, C : grain) :-
        eats(A,B),
        eats(B,C).
```

Listing 2: Godel's typed solution to Shubert's Steamroller